

UNITED STATES PATENT APPLICATION

**METHOD AND APPARATUS FOR LOCAL SYNCHRONIZATION
IN A VECTOR PROCESSOR SYSTEM**

INVENTORS:

Steven L. Scott Citizenship: USA Residence: Eau Claire, Wisconsin
Post Office Address:

Gregory J. Faanes Citizenship: USA Residence: Eau Claire, Wisconsin
Post Office Address:

Brick Stephenson, Citizenship: USA Residence:
Post Office Address:

William T. Moore, Jr., Citizenship: USA Residence: Elk Mound, Wisconsin
Post Office Address:

James R. Kohn Citizenship: USA Residence: Inver Grove Heights, MN.
Post Office Address: **10165 Adam Avenue, Inver Grove Heights, MN. 55077**

Schwegman, Lundberg, Woessner, & Kluth, P.A.

1600 TCF Tower

121 South Eighth Street

Minneapolis, Minnesota 55402

Attorney Docket 01376.733us1

1
2 **MULTISTREAM PROCESSING MEMORY- AND BARRIER-**
3 **SYNCHRONIZATION METHOD AND APPARATUS**
4
5

6 **Field of the Invention**

7 This invention relates to the field of computers, and more specifically to a
8 method and apparatus to support multistream processing with combined memory-
9 and barrier-synchronization operations and/or instructions, for example in a vector
10 supercomputer having multiple nodes, each node having a local memory and a
11 plurality of processors.
12

13 **Background of the Invention**

14 As processors run at faster speeds, memory latency on accesses to memory
15 looms as a large problem. Commercially available microprocessors have addressed
16 this problem by decoupling address computation of a memory reference from the
17 memory reference itself. In addition, the processors decouple memory references
18 from execution based on those references.

19 The memory latency problem is even more critical when it comes to vector
20 processing. Vector processors often transfer large amounts of data between memory
21 and processor. In addition, each vector processing node typically has two or more
22 processing units. One of the units is typically a scalar unit. Another unit is a vector
23 execution unit. In the past, the scalar, vector load/store and vector execution units
24 were coupled together in order to avoid memory conflicts between the units. It has
25 been, therefore, difficult to extend the decoupling mechanisms of the commercially
26 available microprocessors to vector processing computers.

27 As multiple parallel processors are used to simultaneously work on a single
28 problem, there is a need to communicate various status between processors. For
29 example, status that each processor has reached a certain point in its processing
30 (generally called a barrier synchronization, since no processor is allowed to proceed

1 beyond the synchronization point until all processors have reached the
2 synchronization point). See, for example, U.S. Patent 5,721,921, which issued
3 2/24/1998 entitled BARRIER AND EUREKA SYNCHRONIZATION
4 ARCHITECTURE FOR MULTIPROCESSORS, which is incorporated in its entirety
5 by reference. For another example, status that various memory operations specified
6 before that point have completed and various memory operations after that point can
7 rely on the fact that they have completed (generally called a memory
8 synchronization).

9 What is needed is a system and method for hiding memory latency in a vector
10 processor that limits the coupling between the scalar, vector load/store and vector
11 execution units. Further what is needed is a fast, repeatable, and accurate way
12 synchronizing operations within a processor and across processors.

13 14 **Summary of the Invention**

15 The present invention provides a method and apparatus to provide specifiable
16 ordering between and among vector and scalar operations within a single streaming
17 processor (SSP) via a local synchronization (Lsync) instruction that operates within a
18 relaxed memory consistency model. Various aspects of that relaxed memory
19 consistency model are described. Further, a combined memory synchronization and
20 barrier synchronization (Msync) for a multistreaming processor (MSP) system is
21 described. Also, a global synchronization (Gsync) instruction provides
22 synchronization even outside a single MSP system is described. Advantageously, the
23 pipeline or queue of pending memory requests does not need to be drained before the
24 synchronization operation, nor is it required to refrain from determining addresses for
25 and inserting subsequent memory accesses into the pipeline.

26 27 **Brief Description of the Drawings**

28 Fig. 1A illustrates a vector processing computer according to the present
29 invention.

1 Fig. 1B illustrates an alternate embodiment of a vector processing computer
2 according to the present invention.
3 Fig. 1C shows a block diagram of an MSP 102 of some embodiments of the
4 present invention.
5 Fig. 1D shows a block diagram of a node 106 of some embodiments of the present
6 invention.
7 Fig. 1E shows a block diagram of a system 108 of some embodiments of the
8 present invention.
9 Fig. 1G illustrates a processor which could be used in the node of Fig. 1D.
10 Fig. 1H is a more detailed diagram of a processor that could be used in the node of
11 Fig. 1D.
12 Fig. 1I shows instruction flow for some vector instructions according to the
13 invention.
14 Fig. 1J shows instruction flow for some vector instructions according to the
15 invention.
16 Fig. 1K shows instruction flow for some vector instructions according to the
17 invention.
18 Fig. 1L shows a block diagram of a P chip/circuit 100 of some embodiments of
19 the present invention.
20 Fig. 2 shows a block diagram of a system 200 of some embodiments of the
21 invention.
22 Fig. 3 shows a block diagram of VU 113 of some embodiments.
23 Fig. 4 shows an M/Gsync logic block diagram 400 of some embodiments.
24 Fig. 5 shows a block diagram of a system 508 of some embodiments of the
25 invention.

26
27

Description of the Preferred Embodiments

28 In the following detailed description of the preferred embodiments, reference
29 is made to the accompanying drawings that form a part hereof, and in which are
30

1 shown by way of illustration specific embodiments in which the invention may be
2 practiced. It is understood that other embodiments may be utilized and structural
3 changes may be made without departing from the scope of the present invention.

4 The leading digit(s) of reference numbers appearing in the Figures generally
5 corresponds to the Figure number in which that component is first introduced, such
6 that the same reference number is used throughout to refer to an identical component
7 which appears in multiple Figures. Signals and connections may be referred to by
8 the same reference number or label, and the actual meaning will be clear from its use
9 in the context of the description.

10 The coupling together of the various units of a vector processor has made it
11 difficult to extend the decoupling mechanisms used in commercially available
12 microprocessors to vector processing computers. Standard scalar processors often
13 decouple the address computation of scalar memory references and also decouple
14 scalar memory references from scalar execution. The MIPS R10K processor is one
15 example of this approach. A method of extending these concepts to a vector
16 processor is discussed below.

17
18 A vector-processing computer 10 is shown in Figure 1A. Vector processing
19 computer 10 includes a scalar processing unit 12, a vector processing unit 14 and a
20 memory 16. In the example shown, scalar processing unit 12 and vector processing
21 unit 14 are connected to memory 16 across an interconnect network 18. In some
22 embodiments, vector processing unit 14 includes a vector execution unit 20
23 connected to a vector load/store unit 22. Vector load/store unit 22 handles memory
24 transfers between vector processing unit 14 and memory 16.

25 The vector unit 14 and scalar unit 12 in vector processing computer 10 are
26 decoupled, meaning that scalar unit 12 can run ahead of vector unit 14, resolving
27 control flow and doing address arithmetic. In addition, in some embodiments,
28 computer 10 includes load buffers. Load buffers allow hardware renaming of load
29 register targets, so that multiple loads to the same architectural register may be in
30 flight simultaneously. By pairing vector/scalar unit decoupling with load buffers, the

1 hardware can dynamically unroll loops and get loads started for multiple iterations.
2 This can be done without using extra architectural registers or instruction cache space
3 (as is done with software unrolling and/or software pipelining).

4 In some embodiments, both scalar processing unit 12 and vector processing
5 unit 14 employ memory/execution decoupling. Scalar and vector loads are issued as
6 soon as possible after dispatch. Instructions that depend upon load values are
7 dispatched to queues, where they await the arrival of the load data. Store addresses
8 are computed early (in program order interleaving with the loads), and their
9 addresses saved for later use.

10 In some embodiments, each scalar processing unit 12 is capable of decoding
11 and dispatching one vector instruction (and accompanying scalar operand) per cycle.
12 Instructions are sent in order to the vector processing units 14, and any necessary
13 scalar operands are sent later after the vector instructions have flowed through the
14 scalar unit's integer or floating point pipeline and read the specified registers. Vector
15 instructions are not sent speculatively; that is, the flow control and any previous trap
16 conditions are resolved before sending the instructions to vector processing unit 14.

17 The vector processing unit renames loads only (into the load buffers). Vector
18 operations are queued, awaiting operand availability, and issue in order. No vector
19 operation is issued until all previous vector memory operations are known to have
20 completed without trapping (and as stated above, vector instructions are not even
21 dispatched to the vector unit until all previous scalar instructions are past the trap
22 point). Therefore, vector operations can modify architectural state when they
23 execute; they never have to be rolled back, as do the scalar instructions.

24 In some embodiments, scalar processing unit 12 is designed to allow it to
25 communicate with vector load/store unit 22 and vector execution unit 20
26 asynchronously. This is accomplished by having scalar operand and vector
27 instruction queues between the scalar and vector units. Scalar and vector instructions
28 are dispatched to certain instruction queues depending on the instruction type. Pure
29 scalar instructions are just dispatched to the scalar queues where they are executed
30 out of order. Vector instructions that require scalar operands are dispatched to both

1 vector and scalar instruction queues. These instructions are executed in the scalar
2 unit. They place scalar operands required for vector execution in the scalar operand
3 queues that are between the scalar and vector units. This allows scalar address
4 calculations that are required for vector execution to complete independently of
5 vector execution.

6 The vector processing unit is designed to allow vector load/store instructions
7 to execute decoupled from vector execute unit 20. The vector load/store unit 22
8 issues and executes vector memory references when it has received the instruction
9 and memory operands from scalar processing unit 12. Vector load/store unit 22
10 executes independently from vector execute unit 20 and uses load buffers in vector
11 execute unit 20 as a staging area for memory load data. Vector execute unit 20 issues
12 vector memory and vector operations from instructions that it receives from scalar
13 processing unit 12.

14 When vector execution unit 20 issues a memory load instruction, it pulls the
15 load data from the load buffers that were loaded by vector load/store unit 22. This
16 allows vector execution unit 20 to operate without stalls due to having to wait for
17 load data to return from main memory 16.

18 Vector stores execute in both the vector load/store unit 22 and the vector
19 execute unit 20. The store addresses are generated in the vector load/store unit 22
20 independently of the store data being available. The store addresses are sent to
21 memory 16 without the vector store data. When the store data is generated in vector
22 execute unit 20, the store data is sent to memory 22 where it is paired up with the
23 store address.

24 The current approach allows scalar computation to execute independently
25 of vector computation. This allows the scalar address and operand computation to
26 execute in parallel, or ahead of the vector instruction stream. In addition, this
27 approach allows vector address generation to execute independently of vector
28 execution. Finally, this approach allows vector memory operations to execute ahead
29 of vector execution, thus reducing memory latency.

1 This decoupling of the scalar, vector load/store, and vector execution does,
2 however, make synchronizing of scalar and vector data more complex. Computer 10
3 incorporates specific instructions and hardware that reduce this complexity.
4

5 **Memory Consistency Model**

6 Most microprocessor architectures provide a memory consistency model,
7 which describes the order in which memory references by different processors are
8 observed.

9 The simplest, and "strongest" memory model is sequential consistency. This
10 model states that references by each processor appear to occur in program order, and
11 that the references of all the processors in the machine appear to be interleaved in
12 some global order (it doesn't matter what this order is so much as that no two
13 processors can observe a different order).

14 In a sequentially consistent multiprocessor, processors can synchronize with
15 each other via normal memory references. If one processor writes a data value, and
16 then a flag variable, for example, another processor can spin on the flag variable, and
17 when it sees that change, can then read the data value and be guaranteed to see the
18 value written by the first processor.

19 While this may be convenient for the programmer, it constrains the hardware
20 in ways that can reduce performance. Hardware must in general not allow reads and
21 writes to different memory locations by the same processor be performed out of
22 order.

23 While many multiprocessors do not provide sequential consistency, most at
24 least guarantee that references by a single processor will occur in program order (e.g.:
25 a read to a memory location will observe the last write by the same processor). Even
26 this can be difficult to do in a high-bandwidth vector processor, however.

27 One of the primary advantages of a vector architecture is that the instruction
28 set conveys dependency information that allows for highly parallel execution. Groups
29 of operations can be handed off to sequencers to be executed independently. This
30 distributed nature of a vector processor makes it difficult and performance limiting to

1 coordinate all scalar and vector memory references and ensure program ordering. The
2 X1 memory model is designed with this in mind.

3 X1 provides a very weak memory consistency model by default, but includes
4 a rich suite of memory synchronization instructions for providing stronger ordering.
5 The memory model is broken into two parts: single-processor ordering and multiple-
6 processor ordering.

7 8 **Single-stream processor ordering**

9 The architecture defines minimal guarantees on the ordering of the effects of
10 vector memory reference operations with respect to each other and to scalar memory
11 references. Explicit intra-SSP memory synchronization instructions normally must be
12 used to guarantee memory ordering between multiple vector memory references and
13 between vector and scalar memory references.

14 Define *program order*, a total ordering relation, between *operations* on a
15 single SSP as the sequence observed by an implementation with a simple serial
16 fetch/execute cycle. Each operation is the execution of a single scalar CPU
17 instruction or a member of the sequence of elemental actions initiated by the
18 execution of a vector instruction. We say that operation A is earlier than operation B
19 if A precedes B in the program order.

20 Define *dependence order*, a partial ordering relation, as a subset of program
21 order. Two operations A and B are ordered with respect to dependence (i.e.: B
22 depends upon A) if A is earlier than B and at least one of these conditions holds:

- 23 • A and B are scalar references to the same address.
- 24 • A synchronization instruction executes between memory references A and B
25 in program order that orders A and B.
- 26 • B reads a scalar register or vector register element written by A, and no
27 operation between A and B in program order writes that register or vector
28 register element (*true dependence*).
- 29 • There exists an operation C between A and B in program order that depends
30 upon A and upon which B depends (*transitivity*).

- A and B are elements of the same ordered scatter instruction.
- A and B are elements of a vector store with a stride value of zero.

Note that apparent output dependence and anti-dependence relations do not order operations in this definition of dependence order, for advanced implementations can use dynamic register renaming to avoid output dependence and anti-dependence constraints.

All implementations of this architecture must respect dependence order. A program is said to have *data races* if it performs memory reference operations A and B, not both loads, that reference the same byte of memory but are not ordered with respect to dependence. The apparent ordering of these operations is not guaranteed, and thus the program behavior may be unpredictable. Operations that are ordered by dependence will appear to the SSP issuing them to execute in program order. They will not necessarily appear to other SSPs in the system to execute in the same order.

Multiple SSP ordering

This model applies to memory references performed and observed by multiple SSPs.

The multi-SSP memory ordering model provides only the following rules:

1. Writes to any given word of memory are serialized. That is, for any given program execution, writes to a given location occur in some sequential order and no processor can observe any other order.
2. A write is considered *globally visible* when no SSP can read the value produced by an earlier write in the sequential order of writes to that location.
3. No SSP can read a value written by *another MSP* before that value becomes globally visible.

All other ordering between SSPs and MSPs must be provided by explicit memory synchronization instructions (Msync and Gsync instructions).

Providing stronger memory models

The base memory model as stated above is fairly weak. Stronger consistency models can be obtained using memory ordering instructions.

Release consistency can be provided using the acquire Gsync and release Gsync instructions. Acquire events must use scalar loads and be followed by an acquire Gsync. Release events must use scalar stores and be preceded by a release Gsyncs. Atomic memory operations are considered as both integer loads and stores for this purpose.

Sequential consistency could in theory be provided by inserting Gsync instructions between all memory references. This would lead to unacceptable performance for production computing, however.

Virtual Address Space

The X1 uses byte addresses, with a 64-bit virtual address space. The top two bits (VA63..62) determine the memory region, which dictates the privilege level and type of address translation:

0 = user virtual

1 = kernel virtual

2 = kernel physical (no translation)

3 =reserved

VA61..48 are not implemented, and must be zero.

Distributed programming models such as UPC, shmem() and MPI will generally use certain bits of the virtual address to represent a "processing element" (PE). The hardware supports this model via a remote address translation mechanism that interprets VA47..38 as a virtual node number.

Typically, software will use one MSP as a distributed memory PE, in which case it would place the virtual PE number in VA47..36. If a single P chip were used as a distributed memory PE, the PE number would be placed in VA47..34. If a whole node were used as a distributed memory PE (perhaps with an OpenMP layer running

under the distributed memory model), then the PE number would be placed in VA47..38.

Eight pages sizes are provided, varying from 64 KB to 4 GB (64KB, 1MB, 4MB, 16MB, 64MB, 256MB, 1GB, 4GB).

Physical Address Space

The X1 provides a maximum 46-bit (64 TB) physical memory address. The *node* is the unit over which a single physical page of memory is distributed. Access to all memory within a node is completely uniform for a given processor (equal bandwidth and latency). The node size for the X1 implementation. The physical memory format allows for up to 1024 nodes (4096 MSPs) and 64 GB of physical memory per node.

The X1 defines three parallel, physical address spaces, which are selected by PA47..46.

0 = main memory (MEM)

1 = reserved

2 = memory-mapped registers (MMR)

3 = widget space (IO)

Accesses to the main memory space may or may not be cached, but are always coherent. Accesses to MMR and IO space are never cached. MMR space contains special processor state, such as coherence directories, routing tables, cache and memory ECC, and various system control and status registers. IO space provides access to memory-mapped I/O devices residing on the Crosstalk channels.

The operating system controls access to the various physical address spaces via the virtual-to-physical translations.

Address Translation

X1 supports two forms of address translation:

1 source translation, in which a virtual address is fully translated by a
2 translation lookaside buffer (TLB) on the local P chip to a physical
3 address on an arbitrary node, and
4 remote translation, in which the physical node number is determined by a
5 simple translation of the virtual address Vnode field, and the
6 remaining virtual address is sent to the remote node to be translated
7 into a physical address offset via a remote translation table (RTT).

8 If enabled, remote translation is performed for user addresses when the virtual
9 address Vnode field (VA47..38) does not match the local virtual node number stored
10 in a control register. Remote translation can be disabled in order to run distributed
11 memory applications on a set of non-contiguous nodes, or run uniprocessor/SMP
12 applications that exceed the memory of one node.

13 14 **Source translation**

15 A TLB is a cache of page table entries, which is loaded on demand under
16 software control. Each P chip contains three separate TLBs, for translating
17 instructions, scalar references and vector references, respectively. The three TLBs are
18 independent, and may contain different sets of page table entries. The vector TLB is
19 replicated four times for bandwidth purposes, but will contain identical entries under
20 normal operation.

21 Each TLB contains 256 entries and is eight-way set associative. Each way can
22 be set to its own page size, ranging from 64KB to 4GB. The maximum TLB "reach"
23 is thus 1TB (256 entries \times 4 GB/entry).

24 Translation involves first matching a virtual address against an entry of the
25 TLB, and then using the matching entry, if found, to transform the virtual address
26 into a physical address.

27 If a TLB match is not found, then a precise TLB Miss exception occurs. The
28 miss handler can read the offending VA from a control register and load the TLB
29 with the appropriate entry before resuming the process. It may also determine that a

1 page must be swapped in from disk, or that the process has made an illegal reference
2 and must be terminated.

3 Assuming a valid match, the physical address is formed by replacing the
4 virtual page number in the virtual address (those bits above the page boundary), with
5 the physical page number from the TLB entry. If the physical address is to a remote
6 node, then the cache allocation hint is forced to non-allocate and the reference is not
7 cached (Get/Put semantics).

8 illustrates a source virtual address translation. In this example, the way of the
9 TLB containing the matching entry is set to a page size of 1MB. Thus, bits 19..0 of
10 the virtual and physical addresses are below the page boundary. These bits are passed
11 through, untranslated, from the virtual to physical addresses.

12 All bits of the virtual page number (VA47..20), must match the
13 corresponding bits of the VPN field in the TLB entry. These bits are completely
14 replaced by the corresponding bits of the PPN field from the TLB (all bits above the
15 page boundary). The top two bits of the physical address (47..46) specify the physical
16 address space. This will be 00 for main memory accesses, 10 for MMR accesses or
17 11 for IO space accesses. VA63..48 are all zero.

18 19 **Remote translation**

20 A user's partition is defined by two values stored in a control register.
21 BaseNode specifies the physical node on which the partition starts and NodeLimit
22 specifies the size of the partition. In addition, MyNode specifies the virtual node
23 number on which the local processor resides.

24 The remote translation process is illustrated in . When remote translation is
25 enabled, the hardware treats VA47..38 (the VNode field) as a virtual node number.
26 All useg virtual addresses with a VNode value not matching MyNode are translated
27 remotely.

28 In this case, the address is checked to make sure it does not exceed the user's
29 partition defined by a NodeLimit. If $VA47..38 \geq \text{NodeLimit}$, then an Address Error

1 exception occurs. The physical node number for the address is computed by adding
2 VA47..38 to BaseNode.

3 The virtual address Offset field (VA37..0) is sent to the resulting physical
4 node as a "remote virtual address" (RVA) to complete the translation. The cache
5 allocation is forced to non-allocate and the reference is not cached (Get/Put
6 semantics).

7 RVA requests bypass the Ecache, since they can never be cached. The M
8 chips contain a set of four, 2-bit Global Address Space ID (GASID) registers, one for
9 each of the local MSPs. When the local M chip sends a packet out the network with
10 an RVA, it includes the value of the two bit GASID for the originating MSP. This is
11 used to qualify the remote translation of the RVA at the destination M chip. It allows
12 for up to four applications using remote translation to be sharing memory at a node.

13 At the remote M chip, remote virtual addresses go through a translation to a
14 pure physical address, with a granularity of 16 MB. The two GASID bits, and bits
15 37..24 of the RVA are used to index into a 64K-entry remote translation table (RTT).
16 Each entry of this table contains a valid bit, and a 12-bit value representing PA35..24
17 (the 16 MB physical page frame). This is appended to the lower 24 bits of the RVA
18 to form a 36-bit physical memory address at the remote node.

19 The remote translation mechanism does not support demand paging. If the
20 indexed RTT entry does not contain a valid translation, then the request cannot be
21 satisfied, and the application will suffer a fatal error.

22 Remote translation allows the operating system at each node to independently
23 manage its local memory. When space is needed for an application across all nodes
24 of a user's partition, the physical pages can be allocated at different offsets on each
25 node, even though the virtual offsets at each node will be the same. The BaseNode
26 addition and NodeLimit check allow arbitrary partition sizes and alignments (as
27 opposed to requiring power-of-two partition sizes and/or requiring partitions to start
28 on naturally-aligned boundaries), making it easier to find partitions in which to run a
29 user's distributed memory application.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

Synchronization

AMOs

The NV ISA (New Vector Instruction Set Architecture) includes the following atomic memory operations. All operands are 64-bit integers and all memory addresses are dword-aligned.

Atomic fetch and add - Adds an addend (A_k) to the value in memory location A_j , saving the sum in memory and returning the previous value of the memory location into A_i . The read-modify-write of memory is performed atomically.

$$A_i \leftarrow [A_j]$$
$$[A_j] \leftarrow [A_j] + A_k$$

Atomic fetch and and-exclusive or - ANDs the value of A_i with the value of memory location A_j , then XORs the result with the value of A_k , saving the result in memory and returning the original value of the memory location into A_i .

$$tmp \leftarrow [A_j]$$
$$[A_j] \leftarrow (A_i \text{ and } [A_j]) \text{ xor } A_k$$
$$A_i \leftarrow tmp$$

1. This operation can be used to atomically set or clear individual bits in memory (for lock operations), perform various logical operations on memory (AND, OR, XOR, EQV), or perform byte or halfword stores into memory.
2. Atomic compare and swap - Compares the value of A_i with the value in memory location A_j . If they are equal, stores the value of A_k into the memory location. Returns the original value of memory into A_i .

$$tmp \leftarrow [A_j]$$
$$[A_j] \leftarrow (A_i == [A_j]) ? A_k : [A_j]$$
$$A_i \leftarrow tmp$$

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

1. This operation is similar to a "load-linked/store-conditional" sequence (implemented in the MIPS and Alpha ISAs), and allows the implementation of arbitrary atomic operations on memory. A memory location is first loaded and the value used in an arbitrary way. The result is then conditionally stored back to memory, based on a comparison with the old value. If the value has been changed in the interim, the store fails, and the operation must be repeated.
2. Atomic add - This operation is just like the atomic fetch and add, except that no result is returned. This is useful when the value of memory is not needed, as it avoids reserving a result register.

$$[A_j] \leftarrow [A_j] + A_k$$

Atomic and-exclusive or - This operation is just like the atomic fetch and and-exclusive or, except that no result is returned. This is useful when the value of memory not needed, as it avoids reserving a result register.

$$[A_j] \leftarrow (A_i == [A_j]) ? A_k : [A_j]$$

All atomic memory operations in the X1 are performed at the memory controllers on the M chips. They are thus very efficient when multiple MSPs are performing operations on synchronization variables concurrently (that is, under moderate to heavy contention). Under light contention, when the same processor is likely to access the same synchronization variable with no intervening references from other MSPs, the X1 AMOs will not perform as well as a LL/SC sequence would, which can operate out of the local cache.

Barrier synchronization

Barriers allow fast, global communication that all processors (or some subset of the processors) in a parallel program have reached a certain point in their execution.

1 The X1 provides hardware barrier support in the router ASICs only. Thus, all
2 barrier routines will require a software barrier for the 8 MSPs on a node pair, at least.
3 The barrier libraries can optionally configure and use the hardware barrier
4 mechanism in the routers for the remainder of the barrier tree in systems that contain
5 routers (small systems don't use routers).

6 The initial plans are to implement barrier routines fully in software, without
7 using the hardware barrier mechanism at all. A good, hierarchical software barrier
8 completes in $O(\log N)$ time, and is only a small constant factor longer than a
9 hardware barrier. Implementing the library in software (using AMOs) avoids the
10 complexity of managing the hardware barrier.

11 **Interprocessor interrupts**

12 The X1 provides a mechanism for the operating system to deliver interrupts
13 to remote processors, along with a small amount of extra state.

14 Each single-stream processor has four general-purpose, external interrupt
15 lines, each of which is controlled by a memory-mapped trigger register. Interrupts
16 can be delivered to a remote processor by simply writing to the trigger register for the
17 desired interrupt line. The individual interrupts are internally maskable by the
18 processor.

19 Each trigger register is 64 bits wide, with each bit representing an event flag.
20 The meanings and usage of the various event flags are by software convention. A
21 write to an interrupt trigger register generates an interrupt to the processor on the
22 associated interrupt line and sets any bits in the register that are set in the write
23 operand (that is, the write data is bitwise ORed into the trigger register).

24 Bits in the trigger register remain set until explicitly cleared. Upon interrupt,
25 the processor can read the trigger register (or trigger registers) to determine what
26 event(s) caused the interrupt. Event flags can be cleared by writing to another
27 memory-mapped register.

28 **Sync instructions**

As described below, the NV ISA provides very few memory ordering guarantees by default. Memory ordering, when required, can be provided by using one of the Sync instructions.

Lsyncs provide ordering among vector and scalar references by the same processor. The Lsync_s_v instruction guarantees that previous scalar references complete before subsequent vector references. The Lsync_v_s instruction guarantees that previous vector references complete before subsequent scalar references. Finally, the Lsync_v_v instruction guarantees that previous vector references complete before subsequent vector references. There are also a few other varieties that provide even lighter-weight guarantees.

These instructions are used when the compiler or assembly language programmer either knows of a data dependence, or cannot rule out the possibility of a data dependence between the various classes of memory references.

Msyncs provide ordering among memory references by the processors of an MSP. The Msync instruction is executed independently on multiple processors within the MSP (typically by either all processors, or a pair of processors performing a "producer/consumer" synchronization). The Msync instruction includes a mask indicating which processors are participating, and all participating processors must use the same mask, else hardware will detect the inconsistency and cause an exception.

The regular Msync orders all previous references by all participating processors within the MSP before all subsequent references by all participating processors. It is essentially a memory barrier and control barrier wrapped into one. The vector Msync acts the same, but only applies to vector references. Lastly, the producer Msync is intended to situations in which one processor is producing results for one or more other processors to consume. It doesn't require the producer to wait to see earlier results possibly written by the "consumers."

Msyncs are highly optimized in the X1. Vector store addresses are sent out to the E chips long before the actual data is available; the store data are sent along later. Load requests that occur after an Msync are checked against the earlier store

addresses. If there is no match, the loads are serviced, even before the data for stores occurring before the Msync have been sent to the E chip.

Gsyncs provide ordering among references made by multiple MSPs. They are generally used whenever data is shared (or potentially shared) between MSPs. Like Msyncs, Gsyncs include a mask of participating processors within an MSP, and all participating processors must issue a Gsync with a consistent mask. The regular Gsync prevents any subsequent memory references by the participating processors from occurring until all previous loads have completed and all previous stores have become globally visible.

A Gsync should be used, for example, before performing a synchronization operation (such as releasing a lock) that informs other processors that they can now read this processor's earlier stores. Several variants of Gsync are provided, including versions optimized for lock acquire and lock release events.

Figure 1B is a block diagram of a vector processing MSP (multistreaming processor) 106 used in some embodiments of the invention. Vector processing MSP 106 in Figure 1B includes a processor 100. Typically, processor 100 is one of a plurality of processors connected to a common cache 24 and memory 26, however in some embodiments, such as that shown, only a single streaming processor (SSP) 100 is used. Processor 100 includes a scalar processing unit 12 and a plurality (e.g., two) vector processing units (14.0 and 14.1). Scalar processing unit 12 and the two vector processing units 14 are connected to memory 16 across interconnect network 18. In the embodiment shown, memory 16 is configured as cache 24 and distributed global memory 26. Vector processing units 14 include a vector execution unit 20 connected to a vector load/store unit 22. Vector load/store unit 22 handles memory transfers between vector processing unit 14 and memory 16.

Figure 1C shows a block diagram of a multistreaming processor (MSP) 102 of some embodiments of the present invention. MSP 102 includes a plurality of P chips or P circuits 100 (each representing one single-streaming processor having a

plurality of vector pipelines and a scalar pipeline), each P chip/circuit 100 connected to a plurality of E chips or E circuits 101 (each representing an external cache, synchronization, and memory-interface function). In some embodiments, every P chip/circuit 100 is connected to every E chip/circuit 101. In some embodiments, four P Chips 100 and four E Chips 101 form one MSP 102. Although the P Chip 100 and the E Chips 101 are sometimes described herein as “chips” as representing one embodiment, in other embodiments, they are implemented with a plurality of chips each, or with a single chip containing a plurality of P circuits 100 and/or E circuits 101.

In some embodiments, each scalar processing unit 12 delivers a peak of 0.4 GFLOPS and 0.8 GIPS at the target frequency of 400 MHz. Each processor 100 contains two vector pipes, running at 800 MHz, providing 3.2 GFLOPS for 64-bit operations and 6.4 GFLOPS for 32-bit operations. The MSP 102 thus provides a total of 3.2 GIPS and 12.8/25.6 GFLOPS. Each processor 100 contains a small Dcache used for scalar references only. A two-MB Ecache 24 is shared by all the processors 100 in MSP 102 and used for both scalar and vector data. In one embodiment, each processor 100 and e-circuit 101 of cache 24 are packaged as separate chips (termed the “P” chip and “E” chips, respectively).

In some embodiments, signaling between processor 100 and cache 24 runs at 400 Mb/s on processor-to-cache connection 32. Each processor-to-cache connection 32 shown in Figure 1C uses an incoming 64-bit path for load data and an outgoing 64-bit path for requests and store data. Loads, in some embodiments, can achieve a maximum transfer rate of fifty-one GB/s from cache 24. Stores, in some embodiments, can achieve up to forty-one GB/s for stride-one and twenty-five GB/s for non-unit stride stores.

In some embodiments, global memory 26 is distributed to each MSP 102 as local memory 105. Each E Chip 101 has four ports 34 to M chip 104 (and through M chip 104 to local memory 105 and to network 107). In some embodiments, ports 34 are sixteen data bits in each direction. MSP 102 has a total of 25.6 GB/s load

bandwidth and 12.8-20.5 GB/s store bandwidth (depending upon stride) to local memory.

Figure 1D shows a block diagram of a node 106 of some embodiments of the present invention. In some embodiments, a node 106 is packaged on a single printed-circuit board. Node 106 includes a plurality of MSPs 102 each connected to a plurality of M chips 104, each M-chip 104 controlling one or more sections of memory 105. In some embodiments, each M chip 104 is connected to memory 105 using a plurality of channels (e.g., eight), each channel having a plurality of direct RAMBUS DRAM chips (e.g., four). In some embodiments, each node also includes a plurality of I/O channels 103 used to connect to a local-area network (e.g., one or more gigabit ethernet connections) and/or storage (e.g., disk storage or a storage-area network). Each node 106 also includes one or more network connections that interconnect the memories of a plurality of nodes, in some embodiments.

In some embodiments, each node 106 includes four MSPs 102 and sixteen M chips 104. M chips 104 contain memory controllers, network interfaces and cache coherence directories with their associated protocol engines. In one such embodiment, memory 26 is distributed round-robin by 32-byte cache lines across the sixteen M chips 104 at each node 106. Thus, the M chip for a particular address is selected by bits 8..5 of the physical address.

Each E Chip 101 is responsible for one fourth of the physical address space, determined by bits 5 and 6 of the physical address. A reference to a particular line of memory is sent to the associated E Chip 101 where the Ecache is consulted, and either the line is found in the Ecache or the request is sent on to an M chip. Bits 7 and 8 of the physical address select one of four M chips connected to each E Chip 101.

Each M chip 104 resides in one of sixteen independent slices of the machine, and the interconnection network 107 provides connectivity only between corresponding M chips on different nodes (thus there are sixteen parallel,

1 independent networks). All activity (cache, memory, network) relating to a line of
2 memory stays within the corresponding system slice.

3 Each M chip 104 contains two network ports 44, each 1.6 GB/s peak per
4 direction. This provides a total peak network bandwidth of 51.2 GB/s in and 51.2
5 GB/s out. Single transfers to/from any single remote destination will use only half
6 this bandwidth, as only one of two ports 44 per M chip 104 will be used. Also,
7 contention from the other processors 100 on node 106 must be considered. Lastly,
8 all inter-node data is packetized, resulting in a smaller ratio of sustained to peak than
9 in the local memory subsystem. Protocol overheads vary from 33% (one way, stride-
10 1 reads) to 83% (symmetric, non-unit-stride reads or writes).

11 Each node 106 also contains two I/O controller chips 103 ("I" chips) that
12 provide connectivity between the outside world and network 107 and memory 26. In
13 some embodiments, each "I" chip 103 provides two XIO (a.k.a. Crosstalk) I/O
14 channels 49, with a peak speed bandwidth of 1.2 GB/s full duplex each. The I chips
15 are connected to each other and to the sixteen M chips 104 with enough bandwidth to
16 match the four XIO channels.

17 This partitioning provides low latency and high bandwidth to local memory
18 105. With a local memory size of up to sixteen GB (sixty-four GB, once 1 Gbit
19 chips become available), most single-processor and autotasked codes should run
20 locally, and most references in distributed-memory codes will be satisfied locally as
21 well. Latency to remote memory will depend upon the distance to the remote node,
22 and the level of contention in network 107.

23 In some embodiments, a limited operating system executes on each node,
24 with a Unicos/mk-like layer across nodes 106. The limited OS will provide basic
25 kernel services and management of two direct-attached I/O devices (a disk array and
26 network interface). All other I/O connectivity is provided by a separate host system.
27 In one such embodiment, the host system also provides the user environment (shell,
28 cross compilers, utility programs, etc.), and can be used to run scalar compute
29 applications.
30

1 Figure 1E shows a block diagram of a system 108 of some embodiments of
2 the present invention. System 108 includes a plurality of nodes 106 each connected
3 to a common network 107. In some embodiments, network 107 is also connected to
4 one or more other networks 109.

5
6 Figure 1F shows an embodiment having a torus network. In some
7 embodiments, each system 108 includes a plurality of nodes 106 interconnected with
8 a network 107. In some embodiments, as is shown in Figure 1F, network 107
9 includes high-speed links 40 used to connect nodes 106 either as a hypercube or as a
10 torus. Other approaches for interconnecting nodes 106 could be used as well.

11
12 Figure 1G shows one embodiment of processor 100. In the embodiment
13 shown, processor 100 includes three sections: a scalar section (SS) 112, a vector
14 section (VS) 118 and an Ecache interface unit (EIU) 116. EIU 116 is the
15 cache/network interface; it operates to ensure high bandwidth between the SS/VS
16 and cache 24.

17
18 A more detailed example of some embodiments of processor 100 is shown in
19 Figure 1H. In the embodiment shown in Figure 1H, SS 112 includes a Instruction
20 Fetch Unit (IFU) 198, a Dispatch Unit (DU) 119, an AS Unit (ASU) 84, a Load/Store
21 Unit (LSU) 134, and a Control Unit (CU) 195. AS Unit 84 includes A Unit 90 and S
22 Unit 92. In one such embodiment, SS 112 implements an out-of-order, 2-way issue
23 superscalar processor.

24 In some embodiments, VS 118 implements a two-pipe vector processor
25 capable of executing eight floating-point operations and 4 memory operations per
26 Lclk. VS 118 includes a Vector Dispatch Unit (VDU) 117, a Vector Unit (VU) 113
27 and a Vector Load/Store Unit (VLSU) 111.

28 In some embodiments, SS 112 is a high performance superscalar processor
29 that implements many of today's RISC superscalar features. It can dispatch, in-order,
30 up to two instructions per Lclk, can then execute instructions out-of-order within the

1 various units, and then graduate in-order up to two instructions per Lclk. The SS
2 also implements speculative execution, register renaming, and branch prediction to
3 allow greater out-of-order execution. The SS can predict up to two branch
4 instruction and uses a Branch History Table (BHT), a Jump Target Buffer (JTB), and
5 Jump Return Stack (JRS) to help insure a high branch prediction rate.

6 SS 112 also contains two 64-bit wide register files, A Registers (AR) and S
7 Registers (SR) 58. The ARs are used mainly for address generation. In some
8 embodiments, there are sixty-four logical ARs and 512 physical ARs that are
9 renamed within A Unit 90. The SRs are used for both integer and floating-point
10 operations. In some embodiments, there are sixty-four logical SRs and 512 renamed
11 physical SRs within S Unit 92.

12 As noted above, SS 112 is capable of issuing up to two integer operations per
13 Lclk using the ARs. In addition, SS 112 is capable of issuing one SR instruction per
14 Lclk that can be either integer or floating point. The decoupled SLSU 134 (scalar
15 load/store unit) of the SS 112 can issue, in order, one load or store per Lclk which
16 may then execute out-of-order with respect to previous scalar memory operations.
17 The SS 112 is also able to issue one branch instruction per Lclk, which allows one
18 branch prediction to be resolved per Lclk.

19 The SS 112 includes separate first level of caches for instructions and scalar
20 data. SS's Instruction Cache (Icache) 60 is 16KBytes in size and is two-way set
21 associative. Each Icache line is thirty-two bytes. Data Cache (Dcache) 196 is also
22 16KBytes and two-way set associative with a thirty-two-byte line size. Icache 60 is
23 virtually indexed and virtually tagged while Dcache 196 is virtually indexed and
24 physically tagged. In some embodiments, Dcache 196 is write through.

25 SS 112 supports virtual memory addressing by maintaining precise
26 exceptions for address translation errors on both scalar and vector memory
27 references. Floating-point exceptions and other errors are not precise.

28 VS 118 is a two pipe vector processor that executes operations at the Sclk
29 rate. Control for VS 118 executes at the LClk rate. In some embodiments, VS 118
30 includes three units, VDU 117, VU 113 and VLSU 111, that are all decoupled from

1 SS 112 and are also decoupled from each other. VDU 117 receives instruction from
2 the DU 119 of SS 112 and couples required scalar operands with these vector
3 operations. The VDU 117 then dispatches these vector instructions to the VU 113
4 and VLSU 111. This allows scalar address generation for vector memory references
5 to execute ahead of the vector operations and also permits the VLSU 111 to run
6 ahead of the VU 118. This decoupling, similar to modern RISC CPUs, helps VS 118
7 conceal latency to both Ecache 24 and memory system 26.

8 VU 118 contains the large Vector Register (VR) file 70, which can operate in
9 two widths, 32-bits or 64-bits. There are thirty-two logical and thirty-two physical
10 VRs 70 that have a Maximum Vector Length (MaxVL) of sixty-four elements. No
11 register renaming is performed on the VRs. When performing 32-bit operations, the
12 VRs are 32-bits wide and when executing 64-bit operations the VRs are 64-bits wide.
13 For both widths of operations, the MaxVL of the VRs remains at sixty-four elements.
14 The VU also contains a Vector Carry Register (VC) that is 1-bit wide and MaxVL
15 elements long. This register is used to support extended integer arithmetic.

16 The VU contains two other register sets that are used to control the vector
17 operations of both the VU and VLSU. A Vector Length Register (VL) 72 is a single
18 64-bit register that contains a value between 0 and MaxVL. The VL serves to limit
19 the number of operations a vector instruction performs. The VS 118 also includes
20 eight Vector Mask Registers (VM) 74 which are each 1-bit wide and MaxVL
21 elements long. VMs 74 are used to control vector instructions on a per element basis
22 and can also contain the results of vector comparison operations.

23 VU 113 is capable of issuing, in order, one vector operation per Lclk. The
24 VU contains two copies, one for each vector pipe, of three Functional Groups (FUG
25 s) 76, that are each capable of executing one operation per Sclk (a clock denoting or
26 specifying a given amount of time or frequency). Each FUG 76 includes multiple
27 functional units that share data paths to and from VRs 70. The FUGs 76 contain
28 both integer and floating-point function units. Vector chaining is supported between
29 the FUGs and writes to memory 26 but is not supported for reads from memory 26.

1 Most 32-bit vector operations will be performed at twice the rate of 64-bit vector
2 operations.

3 VLSU 111 performs all memory operations for VS 118 and is decoupled
4 from VU 113 for most operations. This allows VLSU 111 to load vector memory
5 data before it is used in the VU. The VLSU 111 can issue, in order, one instruction
6 per Lclk. The VLSU 111 can generate four memory operations per Lclk. In some
7 embodiments, the four addresses can be for stride loads, gathers, stride stores, or
8 scatters.

9 To support decoupling of VLSU 111 from VU 113, VS 118 also contains a
10 large Load Buffer (LB) 78. In some embodiments, LB 78 is a 512 by 64-bit element
11 buffer. That is, there are eight vector-register-equivalent buffers that can be used as a
12 staging area for vector loads. Memory load data is placed in LB 78 by VLSU 111
13 until VU 113 has determined that no previous memory operations will fault. VU 113
14 then moves the load data from LB 78 into the appropriate VR 70. In some
15 embodiments, VLSU 111 rotates through the eight load buffers 78 as needed. On a
16 Vector Translation Trap, the load buffer contents are ignored and the vector load
17 instructions associated with the data in the load buffers are restarted.

18 VS 118 also supports virtual memory addressing by maintaining precise
19 exception state for vector address translation errors.

20 EIU 116 maintains a high bandwidth between a P Chip 100 and E Chips 101.
21 In some embodiments, EIU 116 is capable of generating a maximum of four memory
22 references per Lclk to the E Chips 101 and can also receive up to four Dwords of
23 data back from the E Chips 101 per Lclk.

24 EIU 116 supplies instructions to Icache 60 that are fetched from E Chips 101.
25 EIU 116 also controls all scalar and vector memory requests that proceed to E Chips
26 101. The EIU contains buffering that allows multiple scalar and vector references to
27 be queued to handle conflicts when multiple references request the same E Chip in a
28 single Lclk.

29 EIU 116 also cooperates with DU 119, SLSU 134 and VLSU 111 to ensure
30 proper local synchronization of scalar and vector references. This requires, in some

embodiments, that EIU 116 have the capability to invalidate vector store addresses that are resident in Dcache 196.

Instruction Flow

All instructions flow through the processor 100 by first being fetched by IFU 198 and then sent to DU 119 for distribution. In the DU, instructions are decoded, renamed and entered in order into Active List (AL) 94 at the same time that they are dispatched to AU 90, SU 92 and VDU 117.

In some embodiments, a sixty-four-entry AL 94 keeps a list of currently executing instructions in program order, maintaining order when needed between decoupled execution units. Instructions can enter AL 94 in a speculative state - either branch speculative or trap speculative or both. A branch speculative instruction is conditioned on a previous branch prediction. An instruction is trap speculative if it or any previous instruction may trap. Speculative instructions may execute, but their execution cannot cause permanent processor state changes while speculative.

Instructions in the AL proceed from speculative to scalar committed, to committed, to graduated. The AL uses two bits to manage instruction commitment and graduation: trap and complete. An instruction that may trap enters the AL with its trap bit set and complete bit clear. The trap bit is later cleared when it is known the instruction will not trap. Both the trap and complete bits are set for a trapping instruction.

A scalar committed instruction is not branch speculative, will not generate a scalar trap, and all previous scalar instructions will not trap. Scalar commitment is needed for the Vector Dispatch Unit (VDU) to dispatch vector instructions to VU and VLSU. Instructions proceed from being scalar committed to committed.

Committed instructions are not branch speculative, will not trap, and all previous instructions will not trap. After commitment, an instruction proceeds to graduation and is removed from the AL. An instruction may only graduate if it and all previous instructions in the AL are marked complete and did not trap. In some

1 embodiments, the AL logic can graduate up to four instructions per LCLK. Scalar
2 shadow registers are freed at graduation and traps are asserted at the graduation
3 point. Instructions cannot be marked complete until it can be removed from the AL.
4 That requires, in some embodiments, at the least that all trap conditions are known,
5 all scalar operands are read, and any scalar result is written. Some vector instructions
6 are marked complete at dispatch, others at vector dispatch, and the remaining vector
7 instructions are marked complete when they pass vector address translation.

8 Scalar instructions are dispatched by DU 119 in program order to AU 90
9 and/or to SU 92. Most scalar instructions in the AU and SU are issued out-of-order.
10 They read the AR or SR, execute the indicated operations, write the AR or SR and
11 send instruction completion notice back to the DU. The DU then marks the
12 instruction complete and can graduate the scalar instruction when it is the oldest
13 instruction in the AL.

14 All scalar memory instructions are dispatched to AU 90. The AU issues the
15 memory instructions in-order with respect to other scalar memory instructions, reads
16 address operands from AR 56 and sends the instruction and operands to SLSU 134.
17 For scalar store operations, the memory instruction is also dispatched to the AU or
18 SU to read the write data from the AR or SR and send this data to SLSU 134.

19 The SLSU 134 performs address translation for the memory operations
20 received from the AU in-order, sends instruction commitment notice back to the DU,
21 executes independent memory operations out-of-order. For scalar loads, when load
22 data is written into the AR or SR, the AU or SU transmits instruction completion
23 notice back to the DU. Scalar store instruction completion is sent by the EIU 116 to
24 the DU when the write data has been sent off to cache 24.

25 Branch instructions are predicted in the IFU before being sent to the DU. The
26 DU dispatches the branch instruction to either the AU or SU. The AU or SU issues
27 the instruction, reads AR or SR, and sends the operand back to the IFU. The IFU
28 determines the actual branch outcome, signals promote or kill to the other units and
29 sends completion notice back to the DU.

1 The IFU supplies a stream of instructions to Dispatch Unit (DU) 119. This
2 stream of instructions can be speculative because the IFU will predict the result of
3 branches before the outcome of the branch is resolved. AU 90 and SU 92
4 communicate with the IFU to resolve any outstanding predicted branches. For
5 branch instructions that are predicted correctly, the branch instruction is simply
6 promoted by the IFU. For branch mispredictions, the IFU will signal a mispredict to
7 the other units, restart fetching instructions at the correct instruction address and
8 supply the proper instruction stream to the DU.

9 Instructions can enter the AL speculated on one or two previous branch
10 predictions. Each instruction is dispatched with a two bit branch mask, one for each
11 branch prediction register. Instructions can be dependent on either or both branch
12 prediction. A new branch prediction is detected at the point in the instruction
13 dispatch stream where a branch mask bit is first asserted. A copy of the AL tail
14 pointer is saved on each new branch prediction. If the branch is later killed, the tail
15 pointer is restored using the saved copy.

16 At most one new prediction can be dispatched per clock cycle. All
17 predictions are either promoted or killed by the Branch Prediction Logic (BPL).
18 There can be two promotions in a LCLK, but only one branch kill. If there are two
19 outstanding branches, a kill of the older prediction implies a kill of the younger
20 branch prediction. The tail will be restored using the saved tail from the older branch
21 prediction, while the saved tail from the younger prediction is discarded.

22 Another responsibility of the AL logic is to maintain memory consistency.
23 The AL logic orders memory references in the SLSU 134 and VLSU, it informs the
24 SLSU 134 when scalar memory references have been committed so references can be
25 sent to memory, and the AL logic communicates with other units to perform memory
26 synchronization instructions. The AL logic uses the AL to perform these tasks.

27 Because of a potential deadlock, no vector store reference can be sent to
28 memory prior to a scalar or vector load that is earlier in program order. Vector load
29 to vector store ordering is done by a port arbiter in EIU 116 while the AL is used to
30 enforce scalar load to vector store ordering.

1 A scalar load will have a reference sent bit set in the AL if the load hits in the
2 Dcache, or when the port arbiter in the EIU acknowledges the scalar load. The AL
3 has a “reference sent” pointer. This pointer is always between the graduation and
4 scalar commit pointers. The reference sent pointer will advance unless the
5 instruction it points to is a scalar load without its reference sent bit set. When the
6 pointer advances past a vector store instruction, it increments a vector store counter
7 in the VLSU. A non-zero count is required for the VLSU 111 to translate addresses
8 for a vector store reference. The vector store counter is decremented when vector
9 memory issue starts translation of addresses for a vector store instruction.

10 Scalar loads are initially dispatched in the may trap state. If the address
11 generation and translation in the SLSU 134 results in either a TLB miss or an address
12 error, the complete bit is set, indicating a trap. If translation passes, the trap bit is
13 cleared. Scalar loads to non-I/O space may be satisfied in the Dcache or make
14 memory references immediately after successful address translation. The instruction
15 will later complete when the memory load data is written to a physical register.
16 Scalar load references may bypass previous scalar stores provided the loads are to
17 different addresses.

18 Prefetch instructions are initially dispatched to the SLSU 134 in the may trap
19 state but never generate a trap. The trap bit is always cleared and complete bit set
20 after address translation in the SLSU 134. If the address translation was
21 unsuccessful, the prefetch instruction acts as a NOP, making no Dcache allocation or
22 memory reference.

23 Vector instructions are dispatched in-order from DU 119 to VDU 117. VDU
24 117 dispatches vector instructions to both VU 113 and VLSU 111 in two steps.
25 First, all vector instructions are vpredispatched in-order in the VDU after all previous
26 instructions are scalar committed. The VDU 117 separates the stream of vector
27 instructions into two groups of vector instructions, the VU instructions and VLSU
28 instructions. All vector instructions are sent to the VU 113, but only vector memory
29 instructions and instructions that write VL 124 and VM 126 sent to the VLSU 111.

1 Figure 1I shows instruction flow for some vector instructions according to the
2 invention. Vector instructions that are not vector memory instructions or require no
3 scalar operands are marked complete by the DU at dispatch. An example of the
4 instruction flow of such a vector instruction is shown in Figure 1I. This instruction
5 class can then graduate from the active list before the VDU has vdispatched them to
6 the VU.

7
8
9 Figure 1J shows instruction flow for some vector instructions according to the
10 invention. For vector instructions that require scalar operands, the DU also
11 dispatches the vector instructions to the AU and/or SU. An example of the
12 instruction flow of such a vector instruction is shown in Figure 1J. The vector
13 instruction is issued out-of-order in the AU and/or SU, then reads AR and/or SR, and
14 finally sends the scalar operand(s) to the VDU where it is paired with the vector
15 instruction. VDU 117 then vdispatches the instruction and data in-order to the VU.
16 Vector instructions that are not memory instructions but do require scalar operand(s),
17 are marked complete in the AL when the scalar operand(s) have been read in the AU
18 and/or SU.

19 Vdispatch and vlstdispatch are decoupled from each other. For vector
20 instructions that are forwarded to the VLSU, the VDU collects all required scalar
21 operands and vlstdispatches the instruction and data in-order to the VLSU.

22
23 Figure 1K shows instruction flow for some vector instructions according to
24 the invention. Vector memory instructions are dispatched to both the VU 113 and
25 VLSU 111. An example of the instruction flow of such a vector instruction is shown
26 in Fig 1K. Vector memory operations execute in the VLSU 111 decoupled from the
27 VU 113 and send vector memory instruction completion notices back to the DU after
28 vector address translation.

29 If a vector memory instruction results in a translation fault, VLSU 111 asserts
30 a Vector Translation Trap. The instruction in DU 119 reaches the commit point but

1 it won't pass the commit point. It stays there and the rest of the instructions move
2 along and all graduate.

3 Instructions are not allowed to modify the permanent architectural state until
4 they graduate. Since the instruction that caused the translation fault is not allowed to
5 graduate, you have a clear indication of where to restart. The operating system
6 patches up the memory and the program gets restarted at that point.

7 Errors in the VU 113 are handled differently. The vector execution unit
8 makes all of its references and all of its instructions in order so, if it has a fault, it just
9 stops.

10 For vector load instructions, the VU 113 issues the vector memory instruction
11 in-order after address translation has completed in the VLSU 111 and all load data
12 has returned from the memory system. Next, the VU 113 reads data from the LB and
13 writes the elements in the VR.

14 For vector store instructions, the VU 113 waits for completion of address
15 translation in the VLSU, issues the store instruction, and reads the store data from the
16 VR.

17 For non-memory instructions, the VU 113 issues the vector instruction in-
18 order, reads VR(s), executes the operation in the VU's two vector pipes and writes
19 the results into the VR.

20 Instructions are dispatched to AU 90, SU 92 and VDU 117 by DU 119. The
21 DU will dispatch one class of instructions to only one unit, other instructions to two
22 units, and other instruction types to all three units.

23 In some embodiments, DU 119 receives a group of eight instructions from
24 IFU 198 per LCLK. Only one of these instructions can be a branch or jump
25 instruction. Some of these instructions in the group may be invalid.

26 The DU 119 decodes the instruction group, determines register dependencies,
27 renames instructions that require renaming, dispatches up to four instructions per
28 LCLK to the AU, SU, and/or VDU, and then tracks the state of the instructions in the
29 execution units. Outgoing instructions are queued in the AU, SU, and VDU

1 instruction queues. When instructions have completed in the execution units, the DU
2 is informed and the instructions are graduated in order.

3 Decode Logic (DL) in the DU receives up to eight valid instructions from the
4 IFU per LCLK. This instruction group is aligned to a 32-Byte boundary. Some of
5 the instructions may not be valid because of branch and jump instructions within the
6 group of eight instructions. A control transfer to a instruction within a group of
7 eight, will have the instructions before the target instruction invalid.

8 The DL can decode up to 4 instructions per LCLK. These instructions must
9 be naturally aligned on sixteen-byte boundaries. The DL will determine, for each
10 instruction, the instruction attributes, instruction register usage, and instruction
11 destinations for dispatch. The DL will then advance this decoded information to the
12 register renaming logic.

13 Renaming Logic (RL) in the DU accepts up to 4 decoded instructions from
14 the DL per LCLK. The RL determines instruction register dependencies and maps
15 logical registers to physical registers. The AR and SR operand registers of the
16 instructions in the group are matched against the AR and SR result registers of older
17 instructions. This means that the operands must be compared to older instructions
18 within the instruction group and also to older instructions that have already be
19 renamed in earlier Lclk.

20 The RL uses a simpler method for register renaming than true general register
21 renaming. This method is termed Register Shadowing. In register shadowing, AU
22 90 and SU 92 use RAM in their respective units for AR 56 and SR 58, respectfully.
23 The logical AR/SR are mapped to the physical elements of the RAM to allow 8
24 specific physical elements of the RAM per logical AR/SR. This means that logical
25 A5 will be mapped to only one of eight physical registers (A5, A69, A133, A197,
26 A261, A325, A389, A453). Logical A0 and S0 will never be renamed because they
27 are always read as zero.

28 The mapping of logical registers to physical registers for each AR and SR is
29 maintained by a 3-bit user invisible Shadow Pointer (SP) for each logical register
30 (SP_A0-SP_A63, SP_S0-SP_S63). At reset all SPs are cleared to zero.

1 When a AR/SR is used as an operand in a instruction, the SP of that register
2 is used to determine the current physical register. This physical register number is
3 then used as the operand register when the instruction executes in one of the
4 execution units.

5 When an AR/SR is used as a result register, the SP is incremented by one to
6 point to a new physical register. This physical register will then be written to when
7 the instruction executes in one of the execution units. For any instructions after this
8 instruction that use the same logical register as an operand, it will be mapped to this
9 new physical register.

10 To control the renaming of the physical registers, each AR/SR logical register
11 is mapped to a 3-bit user invisible Free Shadow Count (FC) register (FC_A0-
12 FC_A63, FC_S0-FC_S63). At reset the FC registers are set to seven. For each
13 assignment of a new physical register, the associated FC register is decremented by
14 one. Each time an instruction is graduated that assigned a new physical register, the
15 associated FC for that logical register is incremented by one. If the RL tries to
16 rename a logical register that has a FC of zero, the RL holds this instruction and all
17 younger instructions in the group until a physical register is free by graduating an
18 instruction that assigned a physical register to that logical register.

19 The RL also controls the remapping of physical registers to logical registers
20 after a branch mispredict. The RL needs to have the capability of restoring the
21 mapping back to the mapping at the time of the branch instruction that was
22 mispredicted. The RL accomplishes this by maintaining two user invisible 3-bit
23 Branch Count registers (BC0, BC1) for each AR and SR. Each BC is associated with
24 one of the two possible outstanding branch predictions.

25 The BCs count the number of allocations that have been made to a specific
26 logical register since the predicted branch instruction. For each speculative physical
27 register assignment after the speculative branch instruction, the current BC for that
28 speculative branch and logical register is incremented by one. If a second branch is
29 encountered before the first branch is resolved, the other sixty-four BC registers are
30 incremented for register allocation.

1 When the outcome of a speculative branch is resolved, the SP, FC, BC0 and
2 BC1 registers for all ARs and SRs may be modified. For branches that are predicted
3 correctly, the associated BC is cleared to zero. For branches that are predicted
4 incorrectly, the FC registers are incremented by the amount in both the BC0 and BC1
5 registers, the SP registers are decremented by the amount in both the BC0 and BC1
6 registers, and both BC0 and BC1 registers are cleared to zero. This restores the
7 logical to physical mapping to the condition before the speculative branch.

8 The other circumstance where the RL needs to restore the register mapping is
9 when a precise trap is taken. This could be for scalar or vector loads or stores that
10 have a TLB exception. To recover back to the renamed state of the trapping
11 instruction, the SP registers are set equal to the current contents of the SP registers
12 minus seven plus the associated FC register. The FC registers are then set to seven,
13 and the BC0 and BC1 registers are cleared.

14 15 **Vector Unit (VU 113)**

16 Figure 3 shows a block diagram of VU 113 of some embodiments. The VU
17 113 performs the vector integer and floating-point operations in the Vector Section
18 (VS) 118 and one component of vector memory operations. The VU, shown in
19 Figure 3, is a two-pipe vector unit that is decoupled from the Scalar Section (SS 112)
20 and the Vector Load/Store Unit (VLSU 111). The VU 113 receives one vector
21 instruction per LCLK from the Vector Dispatch Unit (VDU 117) in-order and v
22 issues these instructions in-order at a maximum of one instruction per LCLK. For
23 most vector instructions the VU 113 will then read multiple operand elements of the
24 Vector Register (VR 114) file, execute multiple vector operation, and write multiple
25 result elements to the VR. Different vector instructions can operate out-of-order
26 within the VU 113.

27 Vector instructions are received from the VDU and placed in the Vector
28 Instruction Queue (VIQ 189). The VIQ 189 can receive one vector instruction per
29 LCLK from the VDU 117 and can contain up to six valid instructions. If the vector
30 instruction requires scalar operands, the operands are dispatched to VU with the

1 vector instruction. If the vector instruction will write a VR, a Vector Busy Flag
2 (VBFlag) index is also sent with the instruction. The VDU will not vdispatch a
3 vector instruction to VU before all previous scalar instructions have been committed.
4 All vector instructions are sent to the VU 113.

5 The Vector Issue Logic (VIL 310) of the VU reads the first valid vector
6 instruction in the VIQ, resolves all VR conflicts, any functional unit conflicts, and
7 then issues the vector instruction. When a vector instruction is issued, the VIL
8 310 sets the correct VR busy, busies the appropriate functional unit, and clears the
9 indicated VBFlag if needed.

10 Most vector instructions in the VU 113 are executed in the two Vector Pipes
11 (VP0, VP1). Each VP includes, in some embodiments, one copy of the Custom
12 Block (CB) and some additional functions.

13 14 **Vector Registers 114**

15 The thirty-two 64-bit VRs, each 64-elements deep (MaxVL), are split
16 between the two VPs as shown in The description Vector Registers for 64-bit
17 Operations and The description Vector Registers for 32-bit Operations. There are 32
18 logical and 32 physical VRs with no register renaming. Each VP includes, in some
19 embodiments, 32 elements of each VR and are stored in the Custom Register RAM
20 (CRR) of the CB. The 64 elements are split between the VPs by bit 1 of the 6-bit
21 element number. VP0 includes, in some embodiments, vector elements that have bit
22 1 set to 0, and VP1 includes, in some embodiments, vector elements that have bit 1
23 set to 1. When performing 64-bit operations, the VRs are 64-bits wide and each
24 element is stored in one 64-bit CRR element. When performing 32-bit operations,
25 the VRs are 32-bit wide and two consecutive elements in a VP are stored in one
26 64-bit CRR element. The MaxVL for 32-bit operations remains at 64 elements. This
27 vector register organization has two benefits.

28 One, it allows for 64-bit to 32-bit and 32-bit to 64-bit converts to be executed
29 within a VP. Second, it allows 32-bit operations to execute at 4 results per VP per
30 LCLK. Because each VR can have a 64-bit element read and written per SClk, with

two 32-bit elements packed to one 64-bit element of the CRR, reading 32-bit VR operands can operate at 4 elements per VP per LCLK.

Vector Functional Unit Groups 128

Each VP also includes, in some embodiments, the three Functional Unit Groups (FUG1 321, FUG2 322, FUG3c 323). FUGs 124, in some embodiments, are groups of functional units that share common operand and result paths. By sharing paths, only one functional unit in the FUG can be executing at a time.

FUG1 321 includes, in some embodiments, the following vector integer and floating-point functional units: Integer Add/Subtract, Floating-point Add/Subtract, Logical, Integer Compare, and Floating-point Compare.

All FUG1's operations can be 32-bit or 64-bit operations. FUG1's functional units share two operand paths and one result path. Only one 64-bit operation can be performed in the FUG1 per SClk. Two 32-bit operations can be performed per SClk.

FUG2 322 includes, in some embodiments, the following integer and floating-point functional units: Integer Multiply, Floating-point Multiply, and Shift.

All FUG2's operations can be 32-bit or 64-bit operations. FUG2 also has its own two operand paths and one result path for the three functional units and can perform only one 64-bit or two 32-bit operation per SClk.

FUG3c 323 includes, in some embodiments, the following integer and floating-point functional units: Floating-point Divide, Square Root, Leading Zero, Pop Count, Copy Sign, Absolute Value, Logical, Integer Convert, Floating-point Converts, and Merge.

All of FUG3's operations can be 64-bit or 32-bit operations. FUG3c 323 has two operand paths, one result path. All operations but Div/Sqrt can perform one 64-bit or two 32-bit operations per SClk. FUG3c can perform one 64-bit Div/Sqrt operation every fourth LCLK and 32-bit operations once every two LCLK. Both FUG3c for VP0 and VP1 shares one operand path and its result path with two functional units external to the VP0 and VP1.

1 The Bit Matrix Multiply (BMM 325) functional unit and the Iota (IOTA 324)
2 functional unit are external to VP0 and VP1. To provide vector operands to the
3 BMM 325, one of FUG3c's operand paths in each VP is muxed to an external path.
4 The two external operand paths from VP0 and VP1 are connected to the BMM unit.
5 To move results of the BMM to the VR, the result path of FUG3c in both VPs is
6 muxed with the BMM result path. The BMM functional unit is capable of two 64-bit
7 results per LCLK.

8 The IOTA functional unit 324 requires no vector operands but does generate
9 two 64-bit vector results per LCLK. These results are also muxed to FUG3c's result
10 path in each VP.

11 FUG4 326 is a functional unit that performs the following operations on
12 VMs: Fill, Last element set, First Element Set, Pop Count, and Logical.

13 This FUG4 326 does not read the VRs, but performs single operations by
14 reading and writing the VMs. All these operations are MaxVL -bit operations (64).
15 One FUG4 operation can be issued per LCLK.

16 17 **Vector Control**

18 To control the number of vector operations that are performed for each vector
19 instruction, the VU 113 includes, in some embodiments, two other register sets, the
20 Vector Length Register (VL 124) and eight Vector Mask Registers (VM 126).

21 The VL register 124 includes, in some embodiments, a value between 0 and
22 MaxVL and limits the number of vector elements that are read, vector operations that
23 are performed, and vector elements that are written when a vector instructions is
24 issued . The 64-bit VL register is set by an AR operand from the AU and can be
25 read back to an AR.

26 The VM 126 are used to control vector instructions on a per element basis
27 and can also contain the results of vector comparison operations. The eight VMs are
28 each 1-bit wide and MaxVL elements long.

29 Not all vector instructions are performed under the control of VL and VMs.
30 Vector instructions that do execute under the control of VL and/or VM are called

1 elemental vector instructions . The description VL and VM Controlled Vector
2 Instructions shows which vector instructions operate under the control of VL and/or
3 VM.

4 For elemental vector instructions, the current contents of VL is read at vissue
5 to determine the number of operations to perform. At vissue, if VL is currently zero,
6 no vector operations are performed. The only elemental vector operations that are
7 not executed under the control of VL are vi scan(ak,mj) and vi cidx(ak,mj). These
8 two instructions always execute MaxVL elements.

9 Most elemental vector operations also use a VM to control the vector
10 instruction on a per element basis. At vissue, the specified VM is used to control
11 which of the VL elements are written. For VM's elements that are set to one, the
12 indicated operations are performed on the corresponding elements. For VM elements
13 that are cleared, the operations may still be performed but the VR elements will be
14 written, no exception generate, and no data will be written to memory. The BMM vk
15 vector instruction is not controlled by VM.

16 For instructions that are not elemental vector instructions, the number of
17 operations performed is not determine by the contents of VL and VMs. Most non
18 elemental vector operations are memory ordering instructions, single element moves
19 between AR/SRs and the VRs, and VM operations.

20 21 **Vector Operation**

22 VDU 117 controls the vector instruction flow to VU 113 and VLSU 111. No
23 branch speculative or scalar speculative vector instructions are sent to the VU 113 or
24 VLSU 111 from the VDU. Because no scalar speculative vector instructions are sent
25 to the VU 113 and VLSU, the vector units do not have to support a precise trap due
26 to a previous scalar instruction that trapped in SS 112.

27 The VDU performs five functions. The first role of the VDU is to enforce the
28 scalar commitment of previous scalar instructions. The second action of the VDU is
29 to separate the vector instruction stream into two streams of instructions: VU
30 instructions and VLSU instructions. The third role of the VDU is to establish the

1 communication required between the VU 113 and VLSU 111 for vector instructions
2 that are sent to both units and demand coupling. The fourth duty of the VDU is to
3 assemble all scalar operands that are required for a vector instruction. The AU and
4 SU will pass these scalar operands to the VDU. Finally, the VDU dispatches vector
5 instructions to the VU 113 and VLSU. The VDU performs these functions in two
6 stages.

7 The first stage of the VDU is the Vector PreDispatch Logic (VPDL). The
8 VPDL performs the first three functions of the VDU. Vector instructions are
9 dispatched to the Vector PreDispatch Queue (VPDQ) in-order from the Dispatch
10 Unit (DU). The VPDQ can receive two vector instructions per LCLK from DU 119
11 and can contain up to thirty-two vector instructions. Each LCLK, the VPDL attempts
12 to vpredispatch the instruction at the head of the VPDQ to the Vector Dispatch
13 Queue (VDQ) and/or the Vector Load/Store Dispatch Queue (VLSDQ).

14 Before the VPDL can pass the instruction to the VDQ and/or VLSDQ, all
15 previous scalar instructions must be committed. Scalar commitment notice is sent
16 from the Active List (AL) in the Dispatch Unit (DU). Each time the scalar commit
17 pointer moves pass a vector instruction in the AL, the DU sends the VDU a go_vdisp
18 signal that indicates that it is safe to remove a vector instruction from the VPDQ.
19 VDU can receive up to four separate go_vdisp[0-3] signals per LCLK. The VDU
20 maintains a counter that is incremented each time a go_vdisp signal is received. The
21 VPDL determines that the vector instruction at the head of the VPDQ has no
22 previous scalar speculative instructions if this counter is greater than zero. Each time
23 the VPDL removes a instruction from the VPDQ, it decrements this counter.

24 Because the VU 113 and the VLSU 111 are decoupled from each other, there
25 needs to be a method to ensure that the VLSU 111 does not execute instructions that
26 require vector register operands before the last write of that register in the vector
27 instruction stream in VU. To allow for this coupling between the VU 113 and
28 VLSU, the VDU and VU 113 contain three structures.

1 The Vector Busy Flag (VBFlag) registers includes ninety-six 1-bit registers
2 (VBFlag0 - VBFlag95) . One flag is assigned to each vector instruction that writes a
3 vector register. These registers are located in VU 113.

4 At reset, VBFlags are cleared.

5 The Vector MAP (VMAP) structure contains thirty-two 7-bit registers
6 (VMAP0 - VMAP31) that contain the VBFlag register index which was assigned to
7 the last vector instruction in program order that wrote to the corresponding vector
8 register. VMAP0 will have the VBFlag index that is mapped to V0 and VMAP31
9 will contain the VBFlag for V31. The VMAP registers are located in the VDU.

10 At reset, VMAP0 is set to sixty-four, VMAP1 to sixty-five, etc.

11 The VDU also contains the Vector Busy Free Queue (VBFQ) which includes
12 sixty-four 7-bit entries that contain the VBFlags that are currently not allocated to
13 any vector instruction pass the VPDL.

14 At reset, the sixty-four entries are initialized to 0-63.

15 For each vector instruction that writes a VR a VBFlag must be mapped before
16 the VPDL can send the vector instruction to the VDQ and/or the VLSDQ. The
17 VPDL will read the current VMAP entry for Vi, the result register number, push this
18 VBFlag index onto the VBFQ, dequeue the head of the VBFQ, and write this new
19 VBFlag index into the corresponding VMAP entry. The new VBFlag register in VU
20 113 will be marked busy, and the VBFlag index will be passed to the VDQ and/or
21 the VLSDQ with the instruction. When the vector instruction is issued in the VU,
22 the indexed VBFlag will be cleared by the VU. Because the maximum number of
23 vector instructions that can be active after the VPDL is less than ninety-six, there will
24 always be a VBFlag index in the VBFQ.

25 For each vector gather/scatter instruction, the VPDL uses the Vk register
26 number to index into the VMAP registers. The current VBFlag index contained in
27 the VMAP table entry is read and passed with the vector instruction to VLSDQ.
28 When the instruction tries to issue in the VLSU, the VBFlag that the index points to
29 is checked to see if it is still set. If the VBFlag is set, the last write of the vector
30 registers has not executed in the VU, so the instruction will hold issue. If the VBFlag

1 is clear, the instruction is allowed to be issued if there are no other hold issue
2 conditions.

3 Once the VPDL has receive notice from the AL and has mapped the VBFlags,
4 it will vpredispatch the vector instruction to the VDQ and/or the VLSDQ. All vector
5 instructions are vpredispatched to the VDQ but only vector memory instructions and
6 vector instructions that write VL and VM are sent to the VLSDQ.

7 The second stage of the VDU, which performs the last two functions of the
8 VDU, are executed in the Vector Dispatch Logic (VDL) and the Vector Load/Store
9 Dispatch Logic (VLSDL). The VDL and the VLSDL perform the same functions,
10 but execute decoupled from each other.

11 The VDL assembles the required AU and SU scalar operands for each vector
12 instructions bound for VU 113 and then vdispatches these instructions and associated
13 data to the VU. The VLSDL assembles the required AU operands for the vector
14 instructions to be sent to VLSU 111 and vlstdispatches these instructions and required
15 data to the VLSU. The VDL and VLSDL uses seven queues to gather the vector
16 instructions and scalar operands:

17 1) The Vector Dispatch Queue (VDQ) can contain forty-eight valid entries of
18 instructions bound for the VU. The queue can accept one instruction per LCLK from
19 the VPDL and the VDL can remove one instruction per LCLK from VDQ.

20 2) The Vector AJoperand Queue (VAJQ) contains Aj operands that were read
21 in program order in the AU and sent to the VDU. This queue can contain up to
22 thirty-two valid entries and is capable of accepting one operand per LCLK from the
23 AU.

24 3) The Vector AKoperand Queue (VAKQ) contains Ak operands that were
25 read in program order in the AU and sent to the VDU. This queue can contain up to
26 thirty-two valid entries and is capable of accepting one operand per LCLK from the
27 AU.

28 4) The Vector SKoperand Queue (VSKQ) contains Sk operands that were
29 read in program order in the SU and sent to the VDU. This queue can contain up to

thirty-two valid entries and is capable of accepting one operand per LCLK from the SU.

5) The Vector Load/Store Dispatch Queue (VLSDQ) can contain forty-eight valid entries of instructions bound for the VLSU. The queue can accept one instruction per LCLK from the VPDL and the VLSDL can remove one instruction per LCLK from the VLSDQ.

6) The Vector Load/Store AJoperand Queue (VLSAJQ) contains A_j operands that were read in program order in the AU and sent to the VDU. This queue can contain up to thirty-two valid entries and is capable of accepting one operand per LCLK from the AU.

7) The Vector Load/Store AKoperand Queue (VLSAKQ) contains A_k operands that were read in program order in the AU and sent to the VDU. This queue can contain up to thirty-two valid entries and is capable of accepting one operand per LCLK from the AU.

All five operand queues can receive data from the AU or SU speculatively. If a branch was mispredicted, a kill signal will be sent from the IFU and all operand data that has been speculatively written into the operand queues after the mispredicted branch will be removed.

In operation, the VDL dequeues the head of the VDQ and determine if any operands are required for this vector instruction. If operands are required, the VDL attempts to read the head of the indicated operand queues. If all operands are ready, the VDL vdispatches the instruction and data together to the VU 113.

In operation, the VLSDL dequeues the head of the VLSDQ and determine if any operands are required for this vector instruction. If operands are required, the VLSDL attempts to read the head of the indicated operand queues. If all operands are ready, the VLSDL vlstdispatches the instruction and data together to the VLSU.

Order is maintained between the two instructions queues and operand queues by receiving instructions in program order from the VPDL and by also receiving operands from the AU and SU in program order.

VU 113 is a two pipe vector unit that is decoupled from SS112 and VLSU 111. VU 113 receives one vector instruction per LClk from VDU sixty-four in-order and v issues these instructions in-order at a maximum of one instruction per LClk. For most vector instructions the VU 113 will then read multiple operand elements from a VR file 70, execute multiple vector operation, and write multiple result elements to the VR. Different vector instructions can operate out-of-order within VU 113.

Vector instructions are received from VDU 117 and placed in Vector Instruction Queue (VIQ) 81. VIQ 81 can receive one vector instruction per LClk from VDU 117 and can contain up to six valid instructions. If the vector instruction requires scalar operands, the operands are dispatched to VU 113 with the vector instruction. If the vector instruction will write a VR, a Vector Busy Flag (VBFlage) index is also sent with the instruction. VDU 117 will not vdispatch a vector instruction to VU 113 before all previous scalar instructions have been committed. All vector instructions are sent to VU 113.

Vector Issue Logic (VIL) of VU 113 reads the first valid vector instruction in VIQ 81, resolves all VR conflicts, any functional unit conflicts, and then vissues the vector instruction. When a vector instruction is vissued, the VIL sets the correct VR busy, busies the appropriate functional unit, and clears the indicated VBFlag if needed.

Most vector instructions in the VU 113 are executed in the two Vector Pipes (VP0, VP1).

The thirty-two 64-bit VRs, each 64-elements deep (MaxVL), are split between the two VPs. There are thirty-two logical and thirty-two physical VRs with no register renaming. Each VP contains thirty-two elements of each VR 70 and are stored in RAM in a similar manner to that used for SR 58 and AR 56.

The sixty-four elements are split between the VPs by bit 1 of the 6-bit element number. VP0 contains vector elements that have bit 1 set to 0, and VP1 contains vector elements that have bit 1 set to 1. When performing sixty-four-bit operations, the VRs are sixty-four-bits wide and each element is stored in one 64-bit

1 CRR element. When performing 32-bit operations, the VRs are 32-bit wide and two
2 consecutive elements in a VP are stored in one 64-bit CRR element. The MaxVL for
3 32-bit operations remains at sixty-four elements.

4 This vector register organization has two benefits. One, it allows for 64-bit to
5 32-bit and 32-bit to 64-bit converts to be executed within a VP. Second, it allows
6 32-bit operations to execute at 4 results per VP per LClk. Because each VR can have
7 a 64-bit element read and written per SClk, with two 32-bit elements packed to one
8 64-bit element of the CRR, reading 32-bit VR operands can operate at 4 elements per
9 VP per LClk.

10 In the embodiment of processor 100 shown in Figure 1H, each vector pipe
11 includes three functional unit groups (FUG1, FUG2 and FUG3). The functional unit
12 groups share common operand and result paths. Since they share paths, only one
13 functional unit in each FUG can be executing at a time.

14 In some embodiments, FUG1 contains the following vector integer and
15 floating-point functional units: Integer Add/Subtract, Floating-point Add/Subtract,
16 Logical, Integer Compare, and Floating-point Compare. All of FUG1's operations
17 can be either 32-bit or 64-bit operations. FUG1's functional units share two operand
18 paths and one result path. Only one 64-bit operation can be performed in FUG1 per
19 SClk. Two 32-bit operations can be performed per SClk.

20 FUG2 contains the following integer and floating-point functional units:
21 Integer Multiply, Floating-point Multiply, and Shift. All FUG2's operations can be
22 either 32-bit or 64-bit operations. FUG2's functional units also share two operand
23 paths and one result path for the three functional units. It also can perform only one
24 64-bit or two 32-bit operation per SClk.

25 FUG3c contains the following integer and floating-point functional units:
26 Floating-point Divide, Square Root, Leading Zero, Pop Count, Copy Sign, Absolute
27 Value, Logical, Integer Convert, Floating-point Convert, and Merge.

28 All of FUG3's operations can be either 64-bit or 32-bit operations. FUG3c
29 has two operand paths and one result path. All operations but Divide and Square
30 Root can perform one 64-bit or two 32-bit operations per SClk. FUG3c can perform

one 64-bit Divide or Square Root operation every fourth LClks and 32-bit operations once every two LClks. Both FUG3c for VP0 and VP1 shares one operand path and its result path with two functional units external to the VP0 and VP1 (the Bit Matrix Multiply Unit and the Iota Functional Unit).

FUG4 is a functional unit that performs the following operations on VMs: Fill, Last element set, First Element Set, Pop Count, and Logical. This functional unit does not read the VRs, but performs single operations by reading and writing the VMs. All these operations are MaxVL -bit operations (sixty-four). One FUG4 operation can be issued per LClk.

To control the number of vector operations that are performed for each vector instruction, the VU 113 contains two other register sets, the Vector Length Register (VL) 122 and eight Vector Mask Registers (VM) 126.

The VL register 122 contains a value between 0 and MaxVL and limits the number of vector elements that are read, vector operations that are performed, and vector elements that are written when a vector instructions is vissued . The 64-bit VL register 122 is set by an AR operand from the AU and can be read back to an AR.

The Vector Mask Registers are used to control vector instructions on a per element basis. They can also contain the results of vector comparison operations. The eight VMs are each 1-bit wide and MaxVL elements long.

Not all vector instructions are performed under the control of VL 124 and VM 126. Vector instructions that do execute under the control of VL 124 and/or VM 126 are called elemental vector instructions

For elemental vector instructions, the current contents of VL 124 are read at vissue to determine the number of operations to perform. At vissue, if VL 124 is currently zero, no vector operations are performed. The only elemental vector operations that are not executed under the control of VL 124 are vi scan(ak,mj) and vi cidx(ak,mj). These two instructions always execute MaxVL elements.

Most elemental vector operations also use a VM register to control the vector instruction on a per element basis. At vissue, the specified VM is used to control which of the VL elements are written. For VM register elements that are set to one,

1 the indicated operations are performed on the corresponding elements. For VM
2 register elements that are cleared, the operations may still be performed but no VR
3 elements will be written, no exception generate, and no data will be written to
4 memory. The bit matrix multiply (BMM vk) vector instruction is not controlled by
5 VM.

6 For instructions that are not elemental vector instructions, the number of
7 operations performed is not determined by the contents of the VL 124 and VM
8 registers. Most non-elemental vector operations are memory ordering instructions,
9 single element moves between AR/SRs and the VRs, and VM operations.

10 11 **Vector Load/Store Operations**

12 Vector memory load and store instructions perform different than the
13 instructions that are executed in the FUGs. Vector memory operations are executed
14 in both the VU 113 and the VLSU. The VLSU 111 executes decoupled from the
15 VU, generates the addresses for the vector loads and stores, and sends them to EIU
16 116.

17 For vector loads, the VU 113 moves the memory data into the VR. For
18 vector stores, the VU 113 reads a VR and sends the store data to the EIU. The store
19 address and data is sent decoupled to the EIU. This process is described in Patent
20 application No. xx/yyy,yyy, entitled "Decoupled Store Address and Data in a
21 Multiprocessor System", filed on even day herewith, the description of which is
22 incorporated herein by reference.

23 Vector load and gather instructions are executed using the load buffers 78 of
24 the two VPs. Each VP of VU 113 contains a 256 by 64-bit entry load buffer (LB) 78
25 that is loaded with vector memory data by VLSU 111. In some embodiments, LBs
26 78 are separated into eight individual load buffers (LBUF0-LBUF7), each sixty-four
27 64-bit elements deep.

28 When a vector load instruction reaches the head of the VIQ 81, the Vector
29 Issue Logic performs two issue checks. First, the vector result register must be not
30 busy. Second, VLSU 111 must have generated all the vector memory addresses

1 without an address fault, and indicated this to VU 113 by setting the control for one
2 of the eight LBs 78. The VIL will then issue the vector load or gather instruction to
3 the control section of one of the eight LBs. When all the memory load data has been
4 loaded into the LB for this load, the LB control will move the elements from the LB
5 into the VR. Only the elements specified by VL 124 and the VM will be written into
6 the VR. Eight moves from the LB to the VR can occur per LClk. When all of the
7 elements have been removed from the LB, the LB is marked clear by VU 113 so
8 VLSU 111 can reallocate it. If the addresses generated in the VLSU 111 generated a
9 fault, VU traps.

10 Vector store and scatter instructions are also executed in both the VU 113 and
11 VLSU 111. For vector store instructions, the VLSU 111 again generates all of the
12 memory addresses, transmits them to the EIU, and send notice to the VU 113 if the
13 addresses generated a fault. The VU, then waits for the vector store instruction to
14 reach the head of the VIQ 81, checks if the VR to be stored is busy, verifies that the
15 addresses generated in the VLSU 111 did not fault and then send the store data to the
16 EIU for the elements that are indicated by the VL 124 and the VM. Each VP has an
17 operand path from the VR to an external path that is linked to the EIU. This allows
18 for four vector store data operands to be sent per LClk.

19 VLSU 111 will be discussed next. VLSU 111 is the vector address
20 generation engine of processor 100. As note above, execution within VLSU 111 is
21 decoupled from SS 112 and VU 113.

22 VLSU 111 generates addresses for VU 113. The VLSU 111 receives its
23 vector instructions from VDU 117, interfaces with VU 113 for gather and scatter
24 index operands, and transmits memory requests to EIU 116.

25 The VLSU 111 receives one instructions from the VDU per LClk.
26 Instructions are placed in VLSEQ, which can contain up to six valid instructions. The
27 VLSEQ receives any scalar operands needed for the vector instructions from VDU
28 117 with the instruction. The VLSU 111 receives all vector memory instructions and
29 any vector instruction that writes a VL 124 or VM register. (VLSU 111 receives

1 instructions that write VL 124 and VM registers because the VLSU 111 maintains its
2 own private copy of the VL 124 and VM registers.

3 The Vector Load/Store Issue logic executing in VLSU 111 performs all the
4 required checks for the vector memory instructions at the head of the VLSIQ and
5 attempts to issue the instruction to Vector Address Generation logic. The Vector
6 Load/Store Issue logic can issue one instruction per LClk.

7 The Vector Address Generation logic 130 generates four memory requests per
8 LClk, performs four address translations per LClk using the four Vector TLBs
9 (VTLB), and places the four physical addresses in one of four Vector Request
10 Buffers (VRB0-VRB3).

11 As noted above, for vector loads, the VLSU 111 assigns one of eight LBs 78.
12 For strided vector loads, the Vector Load/Store Issue logic only checks for a free LB
13 78 that can be allocated. If an LB 78 is free, it issues the load to the Vector Address
14 Generation logic 130, assigns one of the eight LBs 78 to the vector load and informs
15 VU 113 that the selected LB is valid. For vector gather instructions, the Vector
16 Load/Store Issue logic needs to check for an allocatable LBUF and also that the VR
17 needed for the index is valid and ready. The Vector Load/Store Issue logic uses the
18 VBFlags to check to see that the VR needed for the index is valid and ready.

19 In some embodiments, VU 113 contains ninety-six VBFlags. For each vector
20 instruction that writes a VR 114, VDU 117 assigns a VBFlag index to that
21 instruction and sets the indexed VBFlag busy. When VU 113 issues the vector
22 instruction, it clears this flag.

23 When a gather instruction is sent from VDU 117 to VLSU 111, it includes the
24 last VBFlag index of the vector index (Vk). When the gather instruction reaches the
25 head of the VLSIQ, the Vector Load/Store Issue logic checks to see if the indexed
26 VBFlag in the VU is still set. If it is, the last write of the k-operand has not occurred
27 and the Vector Load/Store Issue logic holds issue. If the VBFlag is cleared, the
28 Vector Load/Store Issue logic 120 then requests the VR from the VU. When the VR
29 is free, (it might still be busy from the instruction that cleared the VBFlag) the VU
30 113 will send the VR elements to the VLSU 111 four elements per LClk. The VU

1 113 reads these elements from the VR and sends them to the VLSU 111 through an
2 external register path. The Vector Load/Store Issue logic also assigns an LB 78 to
3 the gather instruction.

4 In some embodiments, all vector store instructions require verification that all
5 previous scalar and vector loads have been sent to memory. Vector load to store
6 ordering is done by EIU 116 while the scalar load to vector store ordering is enforced
7 by Active List 94 in the DU 119 and by the Vector Load/Store Issue logic 120. The
8 VLSU 111 receives a signal from AL 94 each time the “reference sent” pointer
9 passes a vector store instructions. The VLSU 111 can receive four reference sent
10 signals per LClk. The VLSU 111 increments a vector store counter each time it
11 receives a signal. For the Vector Load/Store Issue logic to issue a vector store
12 instruction, the vector store counter must be non-zero. Each time a vector store
13 instruction is issued, the counter is decremented.

14 For scatter operations, the VLSU 111 conducts the same VBFlag busy check
15 that is performed for gather instructions. When the VBFlag is cleared, the VLSU
16 111 again asks the VU 113 for the VR and the VU 113 send the VLSU 111 four
17 elements per LClk when the register is available.

18 The Vector Address Generation logic of the VLSU 111 receives the base
19 address from the Vector Load/Store Issue logic. For strided references, it also
20 receive the stride value from the VLSIL. For gather and scatter vector instructions it
21 receives the vector indexes from the VU. The Vector Address Generation logic can
22 generate up to four addresses per LClk. The number of addresses generated are
23 determined by the contents of VL 124 and the selected VM. If the VL 124 is
24 currently zero, no vector addresses are generated. For non-zero VL, the Vector
25 Address Generation logic generates VL addresses. If the corresponding element in
26 the selected VM is not set, this address is not passed to VTLB.

27 In some embodiments, VLSU 111 contains four VTLBs (VTLB0 - VTLB3).
28 All vector addresses are translated by the VTLBs and perform the required address
29 check. The VTLB reports completion of translation and any error back to the DU.
30 Possible errors are:

- 1 1) Address Error: unaligned access, or access to a privileged or
- 2 reserved memory region;
- 3 2) VTLB Fault: miss in the TLB;
- 4 3) VTLB Conflict: multiple hits in the TLB; and
- 5 4) VTLB Modify Fault: store to a page marked read only.

6 After VTLB physical addresses are placed in Vector Request Buffers (VRB0-
7 VRB3). Each VRB can contain sixty-four valid addresses.

8 For vector load instructions, the Vector Address Generation logic also creates
9 a Vector Outstanding Request Buffer (VORB) entry for each vector address. The
10 VORB contains 512 entries that is indexed by the transaction ID (TID) sent to the E
11 Chip 101 in the request packet and returned in the response. Each VORB entry
12 contains LB 78 indexes that point to the LB 78 entry that the data should be loaded
13 into when it returns from E Chip 101.

14
15 Figure 1L shows a block diagram of a P chip/circuit 100 of some
16 embodiments of the present invention. In some embodiments, P chip/circuit 100
17 represents one single-streaming processor having one or more vector pipelines
18 supported by a vector load-store unit (VLSU) 111 and one or more scalar pipelines
19 supported by a scalar load-store unit (SLSU) 134.

20 Since various vector operations are internally decoupled (i.e., the addresses
21 and commands to load (or store) elements can be generated out-of-order, rather than
22 sequentially in element order), and various vector and scalar load and store
23 operations are decoupled (i.e., there is no guarantee that loads and stores will
24 complete in the order specified in the program code), there is an occasional need for
25 synchronization instructions that can be inserted into the code that command that the
26 hardware complete some or all memory operations for instructions that issued earlier
27 before proceeding to execute memory operations for instructions that issued later.
28 The present invention provides a plurality of different synchronization instructions
29 that can be inserted into the code to obtain the needed guarantees of ordering.

30 Since the processor generates memory requests in a pipelined manner, sync

1 tags for the synchronization operations can be inserted into early stages of various
2 pipelines involved with the operations, and then processed (synchronized with other
3 sync tags in other pipelines) at later stages, rather than waiting for the pipelines to
4 empty to do the synchronization.

5 As described in detail below, there are three forms of Sync instructions.
6 Local processor syncs (Lsync) provide ordering among the scalar and vector units of
7 a single processor. MSP syncs (Msync) provide memory ordering among the
8 processors of a single Multi-Stream Processor. Global syncs (Gsyncs) provide
9 memory ordering among multiple processors in the machine.

10 11 **Vector Load/Store Unit (VLSU) 111**

12 The Vector Load/Store Unit (VLSU) 111 is the vector address generation
13 engine of the P Chip 100. The VLSU's execution is decoupled from the Scalar
14 Section (SS) 112 and the Vector Unit (VU) 113.

15 The VLSU 111 generates addresses (virtual addresses of memory operands)
16 for load and store instructions for the VU 113. The VLSU 111 receives its vector
17 instructions from the Vector Dispatch Unit (VDU) 117, interfaces with the VU 113
18 for gather and scatter index operands, and transmits memory requests to the E-chip
19 Interface Unit (EIU) 116.

20 The VLSU 111 receives one instructions from the VDU 117 per Lclk 199 (a
21 certain clock that specifies a particular unit of time in the processor). Instructions are
22 enqueued in the Vector Load/Store Instruction Queue (VLSIQ) 110, which can
23 contain up to six valid instructions, in some embodiments. The VLSIQ 110 receives
24 any scalar operands needed for the vector instructions from the VDU 117 with the
25 instruction. The VLSU 111 receives all vector memory instructions and any vector
26 instruction that writes VL (vector length register) 124 or VMs (vector mask registers)
27 126. The VLSU 111 receives instructions that write VL 124 and VMs 126 because
28 the VLSU 111 maintains its own private copy (shown) of VL 124 and VMS 126.

29 The Vector Load/Store Issue Logic (VLSIL) 120 performs all the required
30 checks for the vector memory instructions at the head of the VLSIQ 110 and attempts

1 to issue the instruction to the Vector Address Generation (VAG) 130 (in some
2 embodiments, a plurality of VAGs 131, 132, ... 133 (e.g., four) operate in parallel,
3 each generating an address per Lclk). The VLSIL 120 can issue one instruction per
4 Lclk. In some embodiments, each one or the plurality of VAGs 131-133 has a
5 corresponding vector translation look-aside buffer (VTLB) 136-138, and vector
6 request buffer (VRB) 141-143)

7 In some embodiments, the VAG 130 then generates four memory requests per
8 Lclk, performs four address translations per Lclk using the four Vector TLBs (VTLB
9 136, 137, ... 138), and places the four physical addresses in one of the four Vector
10 Request Buffers (VRB0 - VRB3).

11 For vector loads, the VLSU 111 assigns one of eight Load Buffers (LBUF0 -
12 LBUF7) 139 and informs the DU 119, if any addresses faulted. For vector stores, the
13 VLSU 111 only informs the DU 119 if any addresses faulted.

14 15 **Vector Loads**

16 For strided vector loads (wherein each successive element of the vector
17 register is loaded from a memory location having a specified distance from the
18 memory location of the prior element), the VLSIL 120 only checks for a free LBUF
19 139 that can be allocated. If a LBUF is free, it issues the load to the VAG 130,
20 assigns one of the eight LBUFs 139 to the vector load and informs the VU 113 that
21 the selected LBUF 139 is valid. For vector gather instructions (wherein each
22 successive element of the vector register (VR) is loaded from a memory location
23 having an address specified by an element in another vector register having address
24 offsets from the address in a specified base address register), the VLSIL 120 needs to
25 check for an allocatable LBUF and also that the VR needed for the index (address
26 offset) is valid and ready. The VLSIL 120 uses the VBFlags (vector busy flags) 122
27 for this.

28 The VU 113 includes, in some embodiments, 96 VBFlags 122. For each
29 vector instruction that writes a vector register (VR) 114, the VDU 117 assigns a
30 VBFlag index to that instruction and sets the indexed VBFlag 122 busy. When VU

113 issues the vector instruction it clears this flag. When a vector gather instruction is sent from the VDU 117 to the VLSU 111, it includes the last VBFlag index of the vector index (Vk, or the “k-operand”). When a vector gather instruction reaches the head of the VLSIQ 110, the VLSIL 120 checks to see if the indexed VBFlag 122 is still set. If it is, the last write of the k-operand has not occurred and VLSIL 120 holds issue (waits before issuing the instruction). Once the VBFlag 122 is cleared, VLSIL 120 then requests the VR 114 from the VU 113. When the VR 114 is free, (it might still be busy from the instruction that cleared the VBFlag 122) the VU 113 will send the VR elements to the VLSU 111 four element per Lclk 199. The VU 113 reads these elements from the VR 114 and sends them to the VLSU 111 through an external register path. The VLSIL 120 also assigns an LBUF 139 to the gather instruction.

Vector Stores

In some embodiments, all vector store instructions require, because of a potential deadlock between the P Chip 100 and the E Chip 101, that all previous scalar and vector loads have been sent to memory. Vector load to store ordering is done by the EIU 116 while the scalar load to vector store ordering is enforced by an Active List (AL) in the DU 119, and by the VLSIL 120. The VLSU 111 will receive a signal from the AL each time the “reference sent” pointer passes a vector store instructions. The VLSU 111 can receive four reference sent signals per Lclk 199. The VLSU 111 increments a vector store counter each time it receives a signal. For the VLSIL 120 to issue a vector store instruction, the vector store counter must be non-zero. Each time a vector store instruction is issued, the counter is decremented.

For vector scatter operations (wherein each successive element of the vector register (VR) is stored to a memory location having an address specified by an element in another vector register having address offsets from the address in a specified base address register), the VLSU 111 conducts the same VBFlag busy check that is performed for gather instructions. When the VBFlag 122 is cleared, the VLSU 111 again asks the VU 113 for the VR 114 and the VU 113 sends the VLSU

111 four elements from the VR 114 per LCLK when the register is available.

Vector Address Generation

The VAG 130 of the VLSU 111 receives the base address from the VLSIL 120. For strided references, it also receives the stride value from the VLSIL 120. For gather and scatter vector instructions it receives the vector indexes from the VU 113. The VAG 130 can generate up to four addresses per LCLK. The number of addresses generated are determined by the content of VL (length, or number of elements) 124 and the selected VM 126 (mask to determine which elements are selected or not). If the VL 124 is currently zero, no vector addresses are generated. For non-zero VL 124, the VAG 130 generates VL number of addresses. If the corresponding element in the selected VM is not set, this address is not passed to VTLB 135.

The VLSU 111 includes, in some embodiments, a plurality of VTLBs 136, 137, ... 138, which for some embodiments is four VTLBs (VTLB0 - VTLB3). All vector addresses are translated by the VTLBs 130 and perform the required address check. The VTLB 130 reports completion of translation and any error back to the DU 119. Possible errors are:

Address Error: unaligned access, or access to a privileged or reserved memory region

VTLB Fault: miss in the TLB

VTLB Conflict: multiple hits in the TLB

VTLB Modify Fault: store to a page marked read only.

After VTLB 135, physical addresses are placed in the Vector Request Buffers 142, 142, ... 143 (e.g., in some embodiments, VRB0 - VRB3). Each VRB, in some embodiments, can contain 64 valid addresses.

For vector load instructions, the VAG 130 will also create a Vector Outstanding Request Buffer (VORB) entry for each vector address. The VORB

1 includes, in some embodiments, 512 entries that is indexed by the transaction ID
2 (TID) sent to the E Chip 101 in the request packet and returned in the response. Each
3 VORB entry includes, in some embodiments, LBUF indexes that point to the LBUF
4 entry that the data should be loaded into when it returns from the E Chip 101.

5 In some embodiments, the VLSU 111 does some type of cache line reference
6 for stride one and stride two vector references.

7 All sync instructions are sent to the VLSU 111. The VLSU 111 simply
8 passes all “Lsync” to the EIU 116 through the VRBs 141-143. Msync markers are
9 also passed to the EIU 116 through the VRBs 141-143, but will hold the VLSIL 120
10 from issuing further vector instructions until it receives a Msync-completion marker
11 back from the E Chip 101.

12 13 **Memory Interface Section (MS) / E-chip Interface Unit (EIU) 116**

14 The E-chip Interface Unit 116 provides the Scalar Section 112 and Vector
15 Section 118 a path to and from the plurality of (e.g., four) E Chips 101. It also
16 performs the memory synchronization required between the SLSU 134 and the
17 VLSU 111.

18 The EIU 116 includes, in some embodiments, queueing for vector load/store
19 addresses, vector store data, scalar load/store addresses, and scalar data sent to the E
20 Chips 101. The EIU 116 also includes, in some embodiments, a Port ARBiter
21 (PARB) 191-193 for each of the plurality of E Chips 101. Each arbiter 190 performs
22 the Lsync operations, generates the packets that are sent to the E Chips, and arbitrates
23 between vector addresses, scalar addresses and store data, and vector store data. The
24 PARBs 191-193 also send vector store addresses to the four Cache Controllers (CC0
25 - CC3) 197 to invalidate lines in the Dcache 196 when a store address is sent to an E
26 Chip 101.

27 For data and requests returning from the four E Chips 101, the EIU 116
28 routes the data to the IFU (instruction fetch unit) 198, the CCs 197 in the SLSU 134,
29 or the VLSU 111. For external invalidate requests sent from the E Chips 111, the
30 EIU 116 routes these requests to the CC 197.

P Chip to E Chip Interface

In some embodiments, the Vector Request Buffers 141-143 are sixty-four-element buffers that contain vector load or store addresses generated by the VLSU 111. The EIU 116 includes, in some embodiments, a Address Arbiter 157 that routes these addresses to the Vector Request Port Queues (VRQs) 161-163. Each VRQ 161-163 is associated with one E Chip 101. In some embodiments, Address Arbiter 157 routes the request to the correct VRQ 161-163 based on physical address bits (6..5) of the physical memory address.

The Vector Data Buffers 151-153 are also sixty-four-element buffers, and contain vector store data. Vector store data is sent to the E Chips 101 decoupled from the vector store address (i.e., they could be sent before, at the same time, or, most likely after the corresponding address, but the individual data elements are sent in the same order as the corresponding addresses for those elements). The EIU 116 routes the store data to the Vector Data Queues 171-173 in the same order that vector store addresses were sent to VRQs 161-163. This is ensured, in some embodiments, by capturing flow information when the store addresses were routed to the VRQs 161-163.

The EIU 116 includes, in some embodiments, four scalar queues from the four CCs 197 in the SS 112. The Scalar Queues 181-183 contain scalar addresses and scalar store data that have been sent from the corresponding CC 197.

PARBs 191-193 of the EIU 116 control requests to the corresponding E Chips 101. Each PARB 191-193 arbitrates between vector load addresses, vector store data, and scalar load/store addresses and data. For each LClk, the PARB 191-193 will look at the head of the corresponding VRQ 161-163, VDQ 171-173, and SQ 181-183. In some embodiments, if more than one queue includes a valid request, the PARB 191-193 uses a rotating priority to determine which queue makes a E Chip request. Each PARB 191-193 can send one request packet to its E Chip 101 per LClk.

E Chip to P Chip Interface

The E Chip to P Chip communication is split into four E Chip to P Chip logic blocks (E0toP - E3toP) 194 which are associated with each E Chip 101. When response packets return from the E Chip 101, they are routed to the Instruction Fetch Unit (IFU) 198, the CC 197, the VU 113, or the Control Unit (CU) 195. The Transaction ID (TID) that was sent with each request to the E Chip 101 is also returned in the response packet. The EtoP logic routes the packet to the correct unit, based on the TID for that packet. Instruction cache line requests are returned to the IFU 198, scalar references are sent to the CC 197, vector references are returned to the VU 113 and Msync acknowledgments are sent to the CU 195. The EtoP logic 194 also receives unsolicited cache line invalidates from the E Chip 101. The EtoP logic 194 will recognize these requests from a unique TID and routes these requests to the CCs 197.

Scalar Load/Store Unit (LSU) 134

The scalar load/store unit (LSU) 134, includes, in some embodiments, an address generator and four cache controllers 197. Each cache controller 197 includes, in some embodiments, an interface to one of the four PARB ports 191-193, and the portion of the Dcache 196 associated with that port.

Address generation

The SLSU 134 processes scalar loads, scalar stores, prefetches, syncs, and atomic memory operations. Instructions are received from the Address Unit (AU), and can be processed when the required address operands (Aj and optionally Ak) have arrived.

The address add is performed for prefetches and all aligned loads and stores to generate a virtual address. AMOs (atomic memory operations) and unaligned loads and stores use Aj directly as an address.

All instructions that reference memory (all but the syncs) are then translated

1 by STLB 188 as needed and checked for address errors. The STLB 188 reports
2 completion of translation and any errors back to the Active List.

3 Possible errors include:

4 Address Error: unaligned access, or access to a
5 privileged or reserved memory region
6 STLB Fault: miss in the STLB 188
7 STLB Conflict: multiple hits in the STLB 188
8 STLB Modify Fault: store to a page marked read only.
9 Errors for prefetch instructions are ignored, and the
10 instruction is simply discarded.

11 After the STLB 188, instructions are placed into the Initial Request Queue
12 (IRQ 187). The IRQ 187 includes, in some embodiments, 8 entries, allowing up to 8
13 scalar references to be translated after an "Lsync V,S" instruction before the Lsync
14 completes. While these scalar references can't access the Dcache 196 until the Lsync
15 completes, allowing them to pass translation can permit subsequent vector
16 instructions to be dispatched, which will improve performance for certain loops.

17 From the IRQ 187, instructions are sent to one of the four Cache Controllers
18 (CC0 - CC3) 197, steered by physical address bits 6..5. Sync instructions are
19 broadcast to all four cache controllers 197. A Flow Info Queue maintains steering
20 information for scalar store and AMO data values. Each entry records the type of
21 operand(s) and the port to which they should be steered. Data values arrive in order
22 from each of the address unit (AU) and scalar unit (SU), and are steered accordingly.
23 Ak values are used only for AMOs with two operands.

24 25 **Cache controller operation**

26 Each Cache Controller 197 "x" includes, in some embodiments, the tag, state
27 and data arrays for that portion of the Dcache 196 corresponding to E-Chip port
28 (PARB 191-193) "x" (i.e.: having physical address bits 6..5 = x). It includes two
29 primary request-processing pipelines: the IRQ 187, from the address generation
30 logic, and the Forced Order Queue (FOQ 186), through which requests are routed

1 when they cannot be serviced immediately from the IRQ 187.

2 In some embodiments, physical address bits 12..7 of a request are used as the
3 index into the local Dcache 196 (which includes, in some embodiments, 64 indices
4 times two ways, for a total of 128 cache lines). A request from the IRQ 187
5 simultaneously indexes into the Dcache tag, state and data arrays, and also performs
6 an associative, partial address match with all entries in the FOQ 186. The indices
7 and control information for all FOQ entries are replicated for this purpose in an FOQ
8 index array. No request is allowed to leave the IRQ 187 until it is not branch
9 speculative.

10 Read requests that hit in the Dcache 196 and have no potential matches in the
11 FOQ 186 are serviced immediately. Read and write requests that miss in the Dcache
12 196 and have no potential matches in the FOQ 186 cause the line to be allocated in
13 the Dcache 196. A request packet is sent to the Ecache 24 (in E Chips 101)
14 immediately, and the instruction is placed into the FOQ 186 to await the response
15 from the responding E chip 101.

16 In the case of a Dcache allocation, the state of the line is immediately
17 changed to valid; there is no "pending" state. The simple presence of a request for a
18 given line in the FOQ 186 serves the same purpose as a pending state. A subsequent
19 request to the line that is processed from the IRQ 187 before the newly-allocated line
20 has arrived back from the E chip 101 will detect the matching request in the FOQ
21 186 and will thus not be satisfied from the Dcache 196.

22 If a new request from the IRQ 187 matches the partial address of any valid
23 entry in the FOQ 186, then there is a potential exact address match with that entry,
24 and the request is routed through the FOQ 186 to maintain proper request ordering.
25 All AMOs and I/O space references are sent through the FOQ 186, as they do not
26 allocate Dcache lines and can never be serviced from the Dcache 196.

27 Requests that miss in the Dcache 196 and would otherwise have allocated a
28 Dcache line do not do so when they match an FOQ entry; they are run through the
29 FOQ 186 and passed on to the Ecache 24. Simplifying the handling of this relatively
30 infrequent event significantly simplifies the SLSU 134.

1 The FOQ 186 is logically two separate queues unified in a single structure:
2 one queue for accesses to the Dcache 196, and one queue for accesses to the Ecache
3 24 (see Figure 2) in E Chips 101. Each entry in the FOQ 186 can be marked as
4 accessing the Dcache 196 and/or accessing the Ecache 24. FIFO (first-in, first-out)
5 ordering within each class is preserved. That is, all Dcache requests are kept in order
6 with respect to each other, and all Ecache requests are kept in order with respect to
7 each other. However, Dcache-only requests and Ecache-only requests may be
8 dequeued in a different order than they were enqueued.

9 An FOQ entry that is marked to access both the Ecache 24 and Dcache 196
10 may logically be dequeued from the Ecache queue before being dequeued from the
11 Dcache queue. After doing so, the request will still remain in the FOQ 186, but be
12 marked only as a Dcache request. This might happen, for example, for a write
13 request which is able to send its write through to the Ecache 24, but not yet able to
14 write to the Dcache 196 because a newly allocated Dcache line has not yet returned
15 from memory. In general, we expect the Ecache queue to "run ahead" of the Dcache
16 queue, as the head of the Dcache queue will often be waiting for a response from the
17 E chip 101, whereas requests to the E chip 101 can generally be sent as soon as they
18 are ready.

19 Sync instructions are always sent through the FOQ 186. A marker for an
20 "Lsync S,V" is simply passed on to the E chip port PARB 191-193 after all Ecache
21 requests in front of it have been sent. This marker informs the Echip port arbiter
22 191-193 that previous scalar references from this port have been sent to E Chip 101.
23 A Gsync instruction marker is similarly passed through the FOQ 186 and sent to the
24 E chip port arbiter (PARB 191-193) after all previous scalar references have been
25 sent.

26 Processing an "Lsync V,S" instruction from the IRQ 187 causes the head of
27 the IRQ 187 to block, preventing subsequent scalar references from accessing the
28 Dcache 196 or being sent to the Ecache 24 until all vector references have been sent
29 to the Ecache 24 and all vector writes have caused any necessary invalidations of the
30 Dcache 196. Vector write invalidations are performed using a separate port to the

1 Dcache tags. Once all vector writes before the Sync have been run through the
2 Dcache 196, the SLSU 134 is signalled to unblock the IRQ 187s at each cache
3 controller 197. In the meantime, each cache controller sends its "Lsync V,S" marker
4 through its FOQ 186 and on to the E chip port arbiter (PARB) 191-193.

5 Markers for Msync instructions are also sent through the FOQ 186 and passed
6 to the port arbiter. Processing a regular Msync instruction (not a producer or vector
7 Msync) from the IRQ 187 causes the cache controller to go into Dcache bypass
8 mode. In this mode, reads and writes are forced to miss in the Dcache 196 and are
9 sent to the Ecache 24 following the Msync marker. This causes them to see the
10 results of memory references made before the Msync by other P chips 101
11 participating in the Msync.

12 Once an E chip port arbiter 191-193 has received the Msync marker from the
13 scalar SLSU 134 and the vector VLSU 111, it sends an Msync marker to the
14 associated E Chip 101. Once all participating P chips have sent their Msync markers
15 to an E Chip 101, and E Chip 101 has sent the Dcaches 196 any invalidations from
16 writes before the Msync, the E Chip 101 sends Msync completion markers back to
17 the P chips 100. An Msync completion marker from a given E chip 100 turns off
18 Dcache-bypass mode at the associated cache controller 197. External invalidations
19 received from the Ecache 24 in E Chips 101 are performed using a separate port to
20 the Dcache tags.

21 In some embodiments, while in bypass mode, read requests that would
22 otherwise hit in the Dcache 196 actually cause the corresponding Dcache line to be
23 invalidated. This is done so that a subsequent read that is processed just after Dcache
24 bypass has been turned off will not read a line out of the Dcache 196, effectively
25 reading earlier data than an earlier read request which is heading out to the Ecache 24
26 via the FOQ 186. The Dcache bypass mode can also be permanently turned on via a
27 DiagConfig control register in the event of a bug in the SLSU 134.

28 Prefetch instructions are checked in the STL B 188 for possible discarding.
29 Prefetches that would cause STL B misses, would cause addressing exceptions, or
30 translate with a non-allocate hint or to I/O (input/output) or MMR (memory-mapped)

1 register) space are discarded. Other prefetches are converted into regular allocating
2 loads to A0 or S0.

3 In some cases, a message is sent directly to the E chips 101 (via the port
4 arbiter 191-193) by the IRQ 187 tag/state engine. In two cases, the request is
5 serviced directly from the Dcache 196 and in all others it is placed into the FOQ 186.
6 An entry marked pending will always be marked as a Dcache entry as well. It cannot
7 be dequeued from the head of the Dcache queue until the matching response arrives
8 from the Ecache 24, clearing its pending status.

9 All requests that will receive a response from E allocate an ORB
10 (Outstanding Request Buffer) entry. An ORB includes, in some embodiments, 16
11 entries, and is indexed by the transaction ID (TID) sent to the Ecache 24 in the
12 request packet and returned in the response. Each ORB entry includes, in some
13 embodiments, a request type, load buffer index, A or S register number, Dcache
14 index and way, FOQ index, s/dword (single/double word) flag, and physical address
15 bits 4..2 of the request.

16 The ORB entry specifies what to do with the response when it is received
17 from E Chip 101. A "Read" entry causes the response to be placed into the Dcache
18 196 and the requested word to be sent to a register. A "ReadUC" entry causes the
19 result to be sent only to a register, not written into the Dcache 196. A "Prefetch"
20 entry causes the result to be written to the Dcache 196, but not returned to a register.

21 All scalar requests that allocate an ORB entry expect either a single s/dword
22 (e.g., four or eight bytes, in some embodiments) or a full cache line (32 bytes, in
23 some embodiments) in return. For full cache line requests, the requested word will
24 be returned first, with the rest of the cache line following in increasing address,
25 modulo the cache line size. For both word and cache line requests, the requested
26 s/dword is returned directly to the register from the E port via the load buffers
27 associated with the respective scalar section or vector section. Thus, the FOQ entry
28 for a load that caused a Dcache allocation is marked as a dummy. It exists solely so
29 that subsequent IRQ requests to the same line will detect a match and not access the
30 pending line before it returns from E Chip 101. When a dummy request is dequeued

1 from the FOQ 186, it is simply discarded; it does not access the Dcache 196.

2 Accesses to the Dcache 196 from the IRQ 187 and from the FOQ 186 are
3 arbitrated by the SLSU 134 so that at most one occurs per LClk across the four cache
4 controllers 197. Thus, at most one Dcache hit occurs per cycle, allowing the result to
5 be routed directly to the A or S registers via an external data path.

6 In some embodiments, the enqueue bandwidth is increased to the small FIFO
7 used to buffer these load results outside the custom blocks used for the scalar section
8 112, to allow two Dcache accesses per LClk. This is because each time a load
9 request goes through the FOQ 186 (except for the one that causes the allocation), it
10 makes two Dcache accesses: one from the IRQ 187 and one from the FOQ 186. If
11 one only allows one Dcache access per LClk, the IRQ access will waste a cycle of
12 load bandwidth.

13 Ecache requests at the head of the FOQ 186 can be dequeued when the
14 following criteria are met:

15 Regular writes, I/O writes, AMO reads (which return data) and
16 writes (which do not) must all be committed and their
17 data available,

18 I/O reads must be committed,

19 Syncs must be committed, or

20 ReadUC and ReadNA requests can be dequeued immediately.

21 Dcache requests at the head of the FOQ 186 can be dequeued when the
22 following criteria are met:

23 Dummy requests can be dequeued after their matching
24 response from E has returned and written the Dcache
25 196,

26 Reads can be dequeued immediately,

27 Allocating writes (those that were marked pending) can be
28 dequeued after their matching response from E has
29 returned and written the Dcache 196, they are
30 committed and their store data is available, or

Other writes must be committed, and their store data available.

Control Unit (CU) 195

The Control Unit (CU) 195 of the Scalar Section includes, in some embodiments, 64 Control Registers (CR), an interface to the A Unit, an interface to the other P Chips 101 in its MSP 102 and the logic to perform all traps and system related functions.

Figure 2 shows a block diagram of a system 200 of some embodiments of the invention. System 200 includes a plurality of processors 100 as described in Figure 1L, each connected to a plurality of E circuits or E Chips 101, which are in turn connected to memory 105. A plurality of MSPs-with-memory 201 are connected to a network 107.

Various types of sync instructions cause corresponding types of sync markers to be inserted into the VRQs 161-163 and SQs 181-183. Thus a sync instruction, such as an Msync (described below), that requires the memory references before the sync instruction to complete (OB references) to include all scalar and vector references, would insert corresponding markers in both a VRQ (e.g., 161) and the corresponding SQ (e.g., 181). Both markers would need to be at the port arbiter 191 of P0 chip 100 in order for it to send one Msync marker to Q0 230 on E0 Chip 101.

In some embodiments, on each E Chip 101 there is a plurality of queues 230, 240, ... 250 corresponding to the number of P Chips 100 (one or more queues per P chip). In some embodiments, as shown, one queue is provided per P Chip 100. In such embodiments, all memory references from a particular processor would be stopped once an Msync marker reached the head of that processor's queue (e.g., as shown in queue 240 for processor P1, where an Msync marker 241 is at the head of queue Q1 240). In other embodiments, the memory is divided into sections, with each processor having a separate queue for each section.

In the example of Figure 2, Msync checking circuit 260 sees the Msync

1 marker 241, but no corresponding Msyncs at the heads of queues Q0 230 or QN 250,
2 and thus stalls (e.g., with a stall signal 248) further memory accesses from queue
3 240. If Q1 240 later fills up with more memory requests or Msync markers, a stall
4 signal 249 stops processor P1 from sending any more requests to Q1 240. Thus, in
5 some embodiments, Q1 240 can accept up to 63 additional commands or requests
6 after its Msync 241 reaches the head of the queue, blocking further processing out of
7 that Q1 240.

8 Memory requests 237 are in Q0 230 and memory requests 247 are in Q1 240.

9 At a later time, QN 250 will have processed vector access request 251, and its
10 Msync marker 252 will reach the head of that queue, and similarly stall further
11 execution of later requests (e.g., 253-257) from that queue.

12 At likely a still later time, Q0 230 will have processed vector access request
13 231, vector access request 232, and scalar access request 233, and its Msync marker
14 234 will reach the head of that queue, and would similarly stall further execution of
15 later requests (e.g., 235-236) from that queue. However, suppose that all three
16 Msyncs 234, 241, and 252 had participation masks (e.g., 4-bits masks in each Msync
17 marker, obtained from the Msync instructions, that indicated which of the four
18 processors 100 are expected to send corresponding Msyncs to their respective queues
19 230-250) that matched and indicated that only these three processors were
20 participating. Then all three Msyncs would have completed the barrier they are part
21 of, and an Msync-complete signal would be sent back to the respective processors,
22 and further memory requests from all queues 230-250 (i.e., the read or write requests
23 235, 236, 242, 243, 244, 245, and 253-257) would be allowed to immediately
24 proceed.

25 In this way, many memory requests can possibly be queued in, e.g., VRB 141,
26 VRQ 161, and Q0 230 with the Msync marker 234 waiting at the head of Q0 230 for
27 the corresponding Msyncs to arrive in the other queues 23-250 that correspond to its
28 participation mask.

29
30 **Sync instruction processing by the hardware**

This section describes the hardware implementation, according to some embodiments, of the various Sync instructions, which involve many different parts of the P Chip 100. In some embodiments, up to three forms of Sync instructions are provided. Local processor syncs (Lsync) provide ordering among the scalar and vector units of a single processor (in some embodiments, P Chip 100). MSP syncs (Msync) provide memory ordering among the processors of a single Multi-Stream Processor 102. Global syncs (Gsyncs) provide memory ordering among multiple processors (in some embodiments, separate P Chips 100) in the same or different MSPs 102 in the system 108. See the section “**Instruction-Set Architecture for the Synchronization Instructions**” below for a more complete description from the software programmer’s point of view.

Lsync V, V

Vector-vector ordering happens, in some embodiments, by default inside the vector unit 113.

The Lsync_v_v instruction is dispatched after scalar commit to the VLSU 111 and the VU 113.

The Lsync_v_v is ignored and discarded at the head of the VIQ 189, which handles issuing instructions that involve non-memory instructions to the vector registers 114.

When the Lsync_v_v reaches the head of the VLSIQ 110, the VLSU 111 normally passes an Lsync_v_v marker through towards the PARB 191-193.

If the SlowLsyncVV bit is set, then the VLSU 111 waits until earlier vector references have cleared the port arbiter before allowing the Lsync_v_v to pass the head of the VLSIQ 110.

The Lsync_v_v marker is ignored and discarded by the PARB.

Lsync Vj, V

Implemented exactly the same as Lsync V, V.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

Lsync Vj, V, el

Implemented exactly the same as Lsync V, V.

Lsync S, V

The Lsync_s_v instruction is dispatched to the SLSU 134 and (after scalar commit) to the VLSU 111.

The VLSU 111 sends the Lsync_s_v marker down the path towards all four port arbiters (PARB) 191-193 in program order with the references. These references arrive at the PARB 191-193 in the vector request port queues (VRQ) 161-163.

When the Lsync_s_v marker reaches the head of an initial request queue (IRQ 187), it is held until it is not branch speculative, and then sent to all four cache controllers (CCs 197).

At each CC 197, the Lsync_s_v marker flows through a forced order queue (FOQ 186) marked as an Ecache entry.

Once the Lsync_s_v marker has been removed from the IRQ 187, subsequent scalar references may be processed out of the IRQ 187, potentially being sent to the PARB 191-193 before the Lsync_s_v marker which is in the FOQ 186.

When the Lsync_s_v marker reaches the head of the FOQ 186 and is committed, it is sent on to the corresponding PARB 191-193 through its scalar queue (SQ) 181-183.

When the Lsync_s_v marker from the CC 197 reaches the head of the SQ 181-183, if the lsync_s_v_flag is clear, then the PARB 191-193 sets the lsync_s_v_flag and removes the Lsync_s_v marker from its SQ 181-183. Following scalar references from the SQ 181-183 can then be sent to its E Chip 101.

If the Lsync_s_v marker reaches the head of the SQ 181-183 and the lsync_s_v_flag is still set from a previous Lsync_s_v, then the marker waits at the head of the SQ 181-183 until the flag is cleared (by an Lsync_s_v marker from the VRQ 161-163), blocking subsequent scalar references.

1 When the Lsync_s_v marker from the VLSU 111 arrives at the head of the
2 VRQ 161-163, if the lsync_s_v_flag is set, then the PARB 191-193 clears the
3 lsync_s_v_flag and removes the Lsync_s_v marker from the VRQ 161-163, allowing
4 following vector requests to proceed to the E Chip 101.

5 If the Lsync_s_v marker from the VLSU 111 reaches the head of the VRQ
6 161-163 and the lsync_s_v_flag is not set, then the marker waits at the head of the
7 VRQ 161-163 for the lsync_s_v_flag to be set (by an Lsync_s_v from the SQ 181-
8 183), blocking subsequent vector requests. Once the flag is set, the marker can
9 proceed as described above.

11 **Lsync V, S**

12 The Lsync_v_s instruction is dispatched to the SLSU 134 and (after scalar
13 commit) to the VLSU 111.

14 The VLSU 111 sends the Lsync_v_s marker down the path to all four port
15 arbiters (PARB) 191-193 in program order with the references. These references
16 arrive at each PARB 191-193 in the vector request port queues (VRQs) 161-163.

17 As vector write requests pass the PARB 191-193, the addresses are sent to the
18 corresponding CC 197, where they are checked against the contents of the Dcache
19 196 and invalidate any matching Dcache entries.

20 When the Lsync_v_s marker from the VLSU 111 reaches the head of the
21 VRQ 161-163, if the lsync_v_s_flag is clear, then the PARB 191-193 sets the
22 lsync_v_s_flag and removes the marker from the VRQ 161-163. Following vector
23 references from the VRQ 161-163 can then be sent to E Chip 101.

24 If the Lsync_v_s marker reaches the head of the VRQ 161-163 and the
25 lsync_v_s_flag is still set from a previous Lsync_v_s, then the marker waits at the
26 head of the VRQ 161-163 until the flag is cleared (by an Lsync_v_s marker from the
27 SQ 181-183), blocking subsequent vector references.

28 When the Lsync_v_s marker in the SLSU 134 reaches the head of the IRQ
29 187, it is held until it is not branch speculative. It is then sent to all four cache
30 controllers (CCs 197), and the IRQ 187 is blocked until vector_reference_complete

1 signals are received from each of the four PARBs 191-193.

2 At each CC the Lsync_v_s marker flows through the forced order queue
3 (FOQ 186) marked as an Ecache entry (sending the marker through the FOQ 186 is
4 not strictly necessary for ordering, but this is the same way it is done for the
5 Lsync_s_v marker).

6 When the Lsync_v_s marker reaches the head of the FOQ 186 and is
7 committed, it is sent on to the corresponding PARB 191-193 through the scalar
8 queue (SQ 181-183).

9 When the Lsync_v_s marker from the CC 197 arrives at the head of the SQ
10 181-183, if the lsync_v_s_flag is set, then the PARB 191-193 clears the
11 lsync_v_s_flag, removes the Lsync_v_s marker from the SQ 181-183, and sends a
12 vector_reference_complete signal to the initial request queues (IRQ 187) in the
13 SLSU 134. This signal follows any previous vector write Dcache invalidates.

14 If the Lsync_v_s marker from PARB 191-193 reaches the head of the SQ
15 181-183 and the lsync_v_s_flag is not set, then the marker waits at the head of the
16 SQ 181-183 for the lsync_v_s_flag to be set (by an Lsync_v_s from the VRQ 161-
17 163), at which time it can proceed as described above.

18 When the vector_reference_complete signals from all four PARBs 191-193
19 have been received at the IRQ 187, the IRQ 187 is re-enabled, allowing subsequent
20 scalar references to proceed.

21 **Lsync Vr, S**

22 Implemented exactly the same as Lsync V, S.

23 **Lsync fp**

24 Main dispatch is held after an Lsync_fp.

25 The control unit (CU 195) holds issue of the Lsync_fp until the active list is
26 empty.

27 The CU 195 then sends the vector unit (VU 113) a quiet request.

28 After the quiet signal is received from the VU 113, dispatch is released.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

Lsync I

Instruction fetch is held after an Lsync_i

The control unit (CU 195) holds issue of the Lsync_i until it is committed.

The Icache is then invalidated and instruction fetch is resumed.

Msync and Gsync processing by the control unit

Msyncs and Gsyncs have a lot of common functionality. For all the Msyncs and Gsyncs (except “Gsync A”, which is described separately below):

The Sync instruction (i.e., for this section, either Msync or Gsync) is dispatched to the SLSU 134, an ASRQ (issues to the control unit), and, when scalar committed, to the VU 113 and VLSU 111. It enters the active list in the Trap Speculative state.

The Sync can be issued by the control unit (CU 195) when:

the Sync is the oldest uncommitted instruction

the CU 195 is not waiting for any Sync Completes from the E Chips 101 for a previous Msync or Gsync (only one Msync or Gsync may be active at a time) and

there are no unacked (unacknowledged) Syncs in the VLSU 111 and less than 4 unacked Syncs in the VU 113.

The CU 195 goes through a “check-in” procedure, as described in The description “**Check-ins for Msyncs and Gsyncs**” below, to determine if the Msync will succeed or trap (with a Sync Error, Sync Timeout or MINT Pending exception).

If the Sync traps, then:

the CU 195 sends the VLSU 111 and the VU 113 a checkin_complete signal with a trap indication,

1 the Sync is marked as trapping in the AL by setting both the
2 complete and trap bits,
3 when the Sync reaches the graduation point, the CU 195 sends
4 a “request quiet” to the VLSU 111 and the VU 113,
5 when the trapping Sync reaches the head of the VLSIQ 110,
6 and VLSU 111 activity has completed, the VLSU 111 sends a “quiet”
7 to the VU 113,
8 when the VU 113 has received the quiet from the VLSU 111,
9 the VU 113 has received the “request quiet” from the CU 195, the
10 trapping Sync has reached the head of the VIQ 189, and all vector
11 activity has completed, the VU 113 sends a “quiet” signal to the CU
12 195,
13 the CU 195 then sends a “kill” to the VLSU 111 and VU 113,
14 which cleans out any following instructions from the vector side, and
15 the CU 195 initiates exception handling as described in The
16 description “**Check-ins for Msyncs and Gsyncs**” below.

17
18 If the Sync checkin completes successfully, then:

19 the Sync is marked committed in the AL by clearing its trap bit
20 (it will later be completed by the SLSU 134),
21 the CU 195 sends the VLSU 111 and the VU 113 a
22 checkin_complete signal without a trap indication, indicating that the
23 Sync has been committed,
24 when the committed Sync reaches the head of the VLSIQ 110,
25 the VLSU 111 sends a Sync acknowledgment to the CU 195 (only one
26 Sync may be committed but unacknowledged by the VLSU 111), and
27 when the committed Sync reaches the head of the VIQ 189,
28 the VU 113 sends a Sync acknowledgment to the CU 195 (at most
29 four Msyncs or one Gsync can be completed but unacknowledged by
30 the VU 113).

1 **Msync**

2 Assuming that a Msync checkin completes successfully:

3
4 When the Msync reaches the head of the VIQ 189 and its checkin_complete
5 signal has been received, it is discarded and the checkin_complete is acknowledged.

6 When the Msync reaches the head of the VLSIQ 110, and its
7 checkin_complete signal has been received, the checkin_complete is acknowledged
8 and an Msync marker is passed on to the port arbiter (PARB 191-193), in order with
9 respect to the vector reference stream, via the vector request queues (VRQ0-3) 161-
10 163.

11 When an Msync marker reaches the head of the initial request queue (IRQ
12 187) in the SLSU 134, it is held until it is not branch speculative, and then sent to all
13 four cache controllers (CCs 197). It also must be held if any of the Dcaches 196 are
14 currently in bypass mode due to an earlier Msync.

15 At each CC 197 the Msync marker flows through the forced order queue
16 (FOQ 186) marked as an Ecache entry.

17 When the Msync marker reaches the head of the FOQ 186 and is committed,
18 it is sent on to the corresponding PARB 191-193 through the scalar queue (SQ 181-
19 183).

20 After removing a regular Msync marker from the IRQ 187, the four Dcaches
21 196 are put into bypass mode, and the IRQ 187 is blocked until the Msync markers
22 have been removed from the FOQ 186 in all four CCs 197. This keeps the Msync
23 ordered with respect to later scalar references going to E Chip 101. Vector Msyns
24 and producer Msyns do not order subsequent scalar references, so they do not put
25 the Dcache 196 into bypass mode, nor do they block the IRQ 187.

26 At each PARB 191-193, Msync markers from the VRQ 161-163 and SQ 181-
27 183 are held until they match, preventing any subsequent vector or scalar references
28 from being sent to the E Chip 101 until both markers have been received.

29 Once the scalar and vector Msync markers have been received, the PARB
30 191-193 sends an Msync marker to E Chip 101, and the PARB 191-193 can then

1 process additional scalar and vector requests.

2 When an E Chip 101 returns a Sync Complete packet, it is sent to the CU
3 195. It is also sent to the corresponding CC 197, where it turns off Dcache bypass
4 mode if bypass is on due to an Msync.

6 **Msync P**

7 Implemented exactly the same as Msync, except that the CCs 197 do not go
8 into bypass mode when the Msync marker is removed from the IRQ 187.

10 **Msync V**

11 Implemented exactly the same as Msync P.

14 **Gsync**

15 Assuming that a Gsync checkin completes successfully:

16
17 After a Gsync is dispatched, main dispatch (i.e., of all instructions) is halted
18 until the Gsync completes.

19 When a Gsync reaches the head of the VIQ 189, and its checkin_complete
20 signal has been received, it is discarded and the checkin_complete is acknowledged.

21 When a Gsync reaches the head of the VLSIQ 110, and its checkin_complete
22 signal has been received, the checkin_complete is acknowledged and a Gsync marker
23 is passed on to the port arbiter (PARB 191-193), in order with respect to the vector
24 reference stream, via the vector request queues (VRQ0-3) 161-163.

25 When a Gsync marker reaches the head of the initial request queue (IRQ 187)
26 in the SLSU 134, it is held until it is not branch speculative, and then sent to all four
27 cache controllers (CCs 197). It also must be held if any of the Dcaches 196 are
28 currently in bypass mode due to an earlier Msync.

29 At each CC the Gsync marker flows through the forced order queue (FOQ
30 186) marked as an Ecache entry.

1 When the Gsync marker reaches the head of the FOQ 186 and is committed,
2 it is sent on to the corresponding PARB 191-193 through the scalar queue (SQ 181-
3 183).

4 After removing a Gsync marker from the IRQ 187, the IRQ 187 is blocked
5 until the Gsync markers have been removed from the FOQ 186 in all four CCs 197.
6 Of course, there won't be any references behind the Gsync anyway, as dispatch was
7 halted.

8 At each PARB 191-193, Gsync markers from the VRQ 161-163 and SQ 181-
9 183 are held until they match.

10 Once the scalar and vector Gsync markers have been received, the PARB
11 191-193 sends one Gsync marker to E Chip 101.

12 When an E Chip 101 returns a Sync-Complete packet, it is sent to the CU
13 195. It is also sent to the corresponding CC, where it turns off Dcache bypass mode
14 if bypass is on due to an Msync (of course, for Gsync, this won't be the case, but the
15 Sync-Complete packet is generic, so the PARB 191-193 sends it to the CCs 197).

16 When the CU 195 has received all four Sync-Complete signals from the E
17 Chips 101, then main dispatch is re-enabled.

19 **Gsync R**

20 Implemented exactly the same as Gsync.

22 **Gsync CPU**

23 Gsync_cpu operates like the Gsync, described in The description “**Gsync**”
24 above, except:

25
26 after a successful check-in, the CU 195 sends a “request quiet”
27 to the VU 113, and

28 the CU 195 waits for the “quiet” to be returned from the VU
29 113 in addition to receiving the Sync Completes from the E Chips 101
30 before re-enabling main dispatch.

Gsync A

The CU 195 handles the Gsync_a as described in The description “**Msync and Gsync processing by the control unit**” above, except:

it is not dispatched to VLSU 111 or VU 113,
checkin_complete signals (with or without trap indications)
are not sent to the VLSU 111 and VU 113, and
issue is not held back by unacked Sync commitments to the
VLSU 111 and VU 113.

Gsync_a operates like the Gsync, described in The description “**Gsync**”
above, except:

when a Gsync_a marker reaches the PARB 191-193 via the
SQ 181-183, it is not held at the PARB 191-193. Instead it is
immediately removed, and a Gsync_a marker (rather than a regular
Gsync marker) is sent to the E chip.

Check-ins for Msyncs and Gsyncs

Msync and Gsync instructions must be issued with consistent masks and
within a software-selectable time window by all processors participating in the Sync.
Hardware is responsible for verifying correct execution via the Sync “check-in”
mechanism described in this section. Throughout this section, we use the term
“Sync” to mean either an Msync or Gsync.

If the processors use inconsistent masks, then the hardware detects this during
check in and generates a Sync Error exception on all processors involved. Similarly,
if one or more participating processors do not check in before the first processor's
Sync timer goes off, then all processors that have issued Syncs will take a Sync
Timeout exception. The MSPs implement a mechanism, described below, to
precisely interrupt all participating processors in the event of a Sync Error or

1 Timeout.

2 In some embodiments, the ISA requires processors participating in a Gsync to
3 all issue the same flavor (release, acquire, etc.), while processors participating in an
4 Msync may use different flavors (producer, vector, etc.). This also implies that
5 processors may not (legally) mix Gsyncs and Msyncs. However, the P Chips 100 are
6 not able to determine whether all processors participating in a Gsync or Msync are
7 issuing the same flavor of Sync instruction. The P chip check-in procedure looks
8 only at the participation masks.

9 While correct program execution is not ensured, the hardware must not lock
10 up if software issues mixed Syncs with matching masks. Each P Chip 100 behaves
11 internally according to the flavor of Sync issued on that P Chip 100. Sync markers
12 sent to E include two mask bits, which indicate whether the Sync should wait for
13 outstanding loads and/or stores to complete (Msyncs wait for neither, acquire Gsyncs
14 wait for loads only and other Gsyncs wait for both loads and stores). The E Chips
15 101 treat Sync operations with mixed types as the “strongest” of the types. That is, if
16 any of the Sync markers is for a regular Gsync, the operation is treated as a Gsync,
17 else if any of the markers is for an acquire Gsync, the operation is treated as an
18 acquire Gsync, else the operation is treated as an Msync.

19 Software running within an MSP 102 must be able to interrupt the other
20 processors 100 of the MSP 102 in response to interrupts or traps that require a
21 context switch. The MINT instruction is used for this purpose. It is implemented in
22 a way very similar to the Sync Timeout mechanism.

23
24 To support Sync instructions, each P Chip 100 has the following I/O signals:
25 a four-bit Sync mask output to each of the other three P Chips
26 100,
27 an MSP interrupt (Mint) line to each of the other three P Chips
28 100,
29 a Freeze line to each of the other three P Chips 100,
30 a four-bit mask input from each of the other three P Chips 100,

an MSP interrupt line from each of the other three P Chips 100, and
a Freeze input line from each of the other three P Chips 100.

Internally, each P Chip 100 maintains:

- a four-bit input mask register (IMR) for each of the four P Chips 100,
- a four-bit participation mask register (PMR) for the current executing M/Gsync instruction, derived from the mask in the M/Gsync instruction, and
- a Sync timer.

Normal operation

Sync instructions are executed by the control unit (CU 195). An M/Gsync instruction may issue as soon as all instructions prior to the Sync are committed (the Sync reaches the commitment point). When a Sync issues, the P Chip 100 converts the 4-bit mask in the instruction to a physical mask (using information in the MSPControl register). The Sync mask must include the current processor, else a Sync Error exception will occur and the Sync will be aborted. Only one Sync may be executing at a time; it must complete before another Sync instruction may issue.

Assuming a valid mask, the P Chip 100 places the physical mask for the executing Sync into the PMR (participation mask register) and sends a copy of this mask to each of the other processors indicated in the mask over the mask output lines. The outgoing masks are asserted until the Sync checkin completes.

Figure 4 shows an M/Gsync logic block diagram 400 of some embodiments. The three incoming mask sets are aligned as shown in Figure 4 (M/Gsync logic block diagram) into the set of four IMRs (input mask registers), using the CPU field of the Config control register. One of the four IMRs corresponds to the local processor, and will be ignored by the matching logic. Whenever a P Chip 100 detects a

1 transition from a zero value to a non-zero value (any of the four bits are set) on the
2 mask input lines from any other P Chip 100, it latches the 4-bit value into the IMR
3 for that processor. The IMRs are cleared when the Sync completes or the
4 corresponding mask inputs drop to zeros.

5 An executing Msync checkin completes as soon as matching masks have
6 been received from all other processors indicated by the PMR (the IMR
7 corresponding to the local P Chip 100 is ignored for this purpose). That is, for each
8 set bit j in the PMR, j for the local processor, IMR j must match the contents of the
9 PMR. At this point, the PMR and the selected IMRs are cleared, the outgoing
10 participation mask values are set to zeros and the Sync is considered committed.

11 Once the Sync checkin completes, the P Chip 100 sends Msync markers or
12 Gsync markers to each of the four E Chips 101, as described in the earlier sections
13 for each Msync and Gsync instruction.

14 Because the Sync instructions must reach the commitment point on all
15 participating P Chips 100, and the P Chips 100 verify that the masks are consistent,
16 the E Chips 101 are guaranteed to receive a consistent Sync marker from all
17 participating processors. E may receive markers belonging to multiple Sync
18 instructions, and must group subsets of Syncs with consistent (matching)
19 participation masks. However, because the P Chips 100 do not perform a new
20 check-in until the previous M/Gsync has been acknowledged from the E Chips 101,
21 markers from different Syncs on an E-Chip 101 are guaranteed to have disjoint
22 participation masks.

23 24 **Sync Mask Mismatch Errors**

25 If an IMR selected by the PMR for a currently executing Sync does not match
26 the PMR, then the program running on the MSP has generated inconsistent Sync
27 masks. In this case, the Sync instruction is aborted, generating a Sync Error
28 exception, and the outgoing participation masks are set to zeros. The IMRs are not
29 cleared, other than by having their senders drop them. The Sync Error will be
30 detected on all P Chips 100 that have issued a Sync and received at least one

1 inconsistent participation mask. Other processors may have received a participation
2 mask for a Sync involved in a Sync Error but not detected the error themselves.
3 These incoming participation masks will be automatically cleared by the hardware,
4 however, when the transmitting processors (which have detected the Sync Error)
5 drop their outgoing mask values.

6 7 **Time-outs**

8 Each processor starts its own Sync timer when it issues a Sync. If a timer
9 goes off, the processor initiates a freeze of the local MSP in order to prevent race
10 conditions and allow all processors to see a consistent view of the active Sync
11 instructions. To do this, the processor asserts its Freeze line to all four processors in
12 the MSP. Multiple processors might do this at about the same time, and that's okay.

13 When a processor sees one of its incoming Freeze lines asserted, it asserts its
14 own Freeze lines to all processors in the MSP. Thus, all processors in the MSP will
15 quickly raise all their Freeze lines. As each processor sees all its Freeze inputs
16 asserted, it knows that the MSP is fully frozen, and that no more Sync mask inputs
17 will arrive.

18 Any Sync input masks that arrive at a processor during the freeze procedure
19 are allowed to take effect, possibly allowing an active Sync to complete. Any Syncs
20 that complete during the freeze procedure clear the associated PMR and IMRs.

21 Once all the Freeze inputs are asserted and any input mask processing has
22 completed, each processor checks its PMR and IMRs. Any processor that has a non-
23 zero PMR (indicating an issued, but incomplete Sync) clears its PMR and IMRs, sets
24 its outgoing participation masks to zero and takes a local "Sync Time-out" interrupt.
25 The Sync instruction traps precisely; earlier active instructions are drained and later
26 instructions are killed.

27 Any processor that has a zero PMR (indicating that it does not have an issued
28 but incomplete Msync) clears its IMRs (they may or may not be non-zero), drops its
29 Freeze outputs and resumes processing. Processors are prohibited from raising their
30 Freeze lines again until all their Freeze inputs return to zero. If the Sync that caused

1 the time-out completes during the freeze procedure, it is possible that no processor
2 will see an incomplete Sync, and thus no processor will take a Sync Time-out
3 interrupt.

4 When a processor determines that a Sync Timeout interrupt will be generated,
5 it attempts to finish the instructions before the trapping Sync in the active list. In the
6 event that there is a completed Sync at a processor after a freeze completes, the
7 completed Sync instruction will graduate rather than trap.

8 9 **Sync interaction with traps and external interrupts**

10 When a processor receives an external interrupt, it stops dispatch, and tries to
11 complete all its currently active instructions before servicing the interrupt. If one of
12 the current instructions traps, then both the trap and the interrupt will be
13 communicated to the processor (but the trap will be reported in the Cause register
14 ExcepCode field, as traps have higher priority than interrupts).

15 In some embodiments, Msyncs and Gsyncs are treated like other instructions
16 that can take a precise trap. When draining the instructions in response to an
17 interrupt, if one of the active instructions is an Msync or Gsync, the processor will
18 wait for the Sync to complete. The Sync may end up timing out, in which case we
19 follow the procedure outlined above, and signal both the interrupt and the Sync
20 Time-out to the processor (but the Sync Time-out will be reported in the Cause
21 register ExcepCode field, as it has higher priority than interrupts).

22 Similarly, if a processor takes a TLB fault or other precise trap, and there is
23 an earlier Sync instruction active, the Sync will either complete or fail. If it fails,
24 only the Sync Time-out exception will be signalled to the processor, as it was earlier
25 in program order than the faulting memory reference.

26 27 **MSP Interrupts (MINTs)**

28 When a processor handles an external interrupt or TLB fault, it may return
29 quickly before the other processors in the MSP even notice. If, however, the OS
30 decides to interrupt the other processors, it issues a MINT instruction. This causes

1 the processor to go through a procedure similar to the Sync Time-out. It asserts both
2 its Freeze and Mint lines to all processors in the MSP. In response to the Freeze
3 signal, the processors all enter the frozen state, as described in The description
4 “Time-outs” above.

5 Once the freeze completes, all processors in the MSP will see the Mint line
6 asserted from the processor that issued the MINT instruction. If another processor
7 performed a MINT at about the same time, then the processors will see the Mint lines
8 asserted from both of the processors. If another processor initiated an Msync time-
9 out at about the same time, this information will be hidden by the MSP interrupt. In
10 any event, the processors will interpret the event as a MINT, rather than as a Sync
11 time-out.

12 In some embodiments, MINTs set IP bit 4 in the Cause control register, and
13 are processed in the same way as other interrupts. MINTs do not cause precise
14 exceptions on the processor issuing them.

15 While attempting to drain the currently active instructions after a MINT, the
16 processor may encounter an Msync or Gsync instruction. In this case, the Sync
17 instruction takes a precise MSP Interrupt Pending exception (rather than attempting
18 to perform a Sync check-in). Although the occurrence of the MINT is still
19 communicated to software via the IP4 bit, the precise exception has higher priority
20 and is reported in the Cause control register ExcepCode field. This will happen even
21 if the Msync or Gsync is already in the middle of a Sync check-in when the MINT
22 occurs. Software will presumably treat an MSP Interrupt Pending exception in the
23 same way as an MSP Interrupt.

24 If another MINT instruction is encountered when draining the active
25 instructions after a MINT occurs, then that new MINT instruction is executed. This
26 will have little effect on the processor encountering the new MINT instruction: it will
27 set IP4 (which is already set), and then continue trying to drain the remaining active
28 instructions. The second MINT could have a more visible effect on other processors.
29 If another processor had already jumped to its exception handler and cleared IP4,
30 then the execution of a second MINT would cause IP4 to be set again on that

1 processor, possibly causing another exception to be taken on that processor after it
2 returned from the first exception handler.

3
4 Figure 5 shows a block diagram of a alternate system 508 of some
5 embodiments of the invention. System 508 includes a plurality of processors 500
6 much like processor 100 described in Figure 1L except that each processor has a
7 separate output path for vector accesses and another path for scalar accesses, each
8 connected to a plurality of E circuits or E Chips 501, which are in turn connected to
9 memory 105.

10 Various types of sync instructions cause corresponding types of sync markers
11 to be inserted into the VQs 531, 541, ... 551 and SQs 532, 542, ... 552. Thus a sync
12 instruction, such as an Msync (described below), that requires the memory references
13 before the sync instruction to complete (OB references) to include all scalar and
14 vector references, would insert corresponding markers in both a VQ (e.g., VQ0 531)
15 and the corresponding SQ (e.g., SQ0 532). Both markers would need to be at the
16 port arbiter 191 of P0 chip 100 in order for it to send one Msync marker to Q0 230
17 on E0 Chip 101. In contrast, an Msync V that only synchronizes vector OB
18 references from vector OA references would only insert a sync marker into the
19 appropriate VQ 551, 552, ... 553.

20 In some embodiments, on each E Chip 501 there is a plurality of queues 530
21 corresponding to the number of P Chips 500 (one or more queues per P chip). In such
22 embodiments, only vector memory references from a particular processor would be
23 stopped once an Msync marker reached the head of that processor's VQ queue (e.g.,
24 as shown in queue 541 for processor P1, where an Msync marker 549 is at the head
25 of queue Q1 541). In some embodiments, the memory 105 is divided into sections
26 521, with each processor having one or more separate queues for each section 521.

1 **Instruction-set Architecture for the Synchronization Instructions for Some** 2 **Embodiments**

4 **Sync, Global Memory (GSYNC)**

6 Table 1

31	26	25	20	19	14	13	8	7	6	5	0
g 000010			i 000000			j		k 000000		t 00	f
6			6			6		6		2	6

8 **Formats:** GSYNC Aj, f = 000000 SCALAR
9 GSYNC Aj, A f = 000010
10 GSYNC Aj, R f = 000011
11 GSYNC Aj, CPU f = 000001

12 **Purpose:**

13 Order loads and stores to shared memory in a multiprocessor.

14 **Description:**

15 The NV memory consistency model (see xxx) provides almost no guarantees
16 on the apparent ordering of memory references issued by multiple single-stream
17 processors. Rather, ordering can be established when desired by use of explicit
18 synchronization instructions.

19 Gsync instructions provide a partial ordering of memory references issued by
20 participating processors of the local MSP, as viewed by all processors in the system.
21 The participating processors are identified by a mask in Aj. All participating
22 processors must issue Gsyncs with consistent masks.

23 Each type of Gsync identifies two sets of memory references by participating
24 processors: ordered before (OB) and ordered after (OA) references. The OB
25 references consist of some or all of the participating processors' memory references

1 that *precede* the Gsync in program order. The OA references consist of some or all
2 of the participating processors' memory references that *follow* the Gsync in program
3 order.

4 The Gsync guarantees that all of the OB references appear to occur before any
5 of the OA references, as observed by any processor in the system. Specifically, all
6 OB load results must be bound (their values determined) and all OB store values
7 must be globally visible, before any OA load value can be bound or any OA store
8 value can be observed by any other processors. The Gsync does *not* provide any
9 ordering for memory references by participating processors that do not fall into either
10 the OB or OA category.

11 For the regular Gsync instruction (GSYNC Aj), the OB references consist of
12 *all* memory references issued by all participating processors before the Gsync
13 instruction, and the OA references consist of *all* memory references issued by all
14 participating processors after the Gsync instruction. Thus, this instruction provides a
15 complete ordering of memory references preceding the Gsync and memory
16 references following the Gsync.

17 For the acquire Gsync instruction (GSYNC Aj, A), the OB references consist
18 of all *scalar* loads issued by all participating processors before the Gsync instruction,
19 and the OA references consist of *all* memory references issued by all participating
20 processors after the Gsync instruction. Thus, this instruction ensures that previous
21 scalar loads have completed before any subsequent references are performed.

22 For the release Gsync instruction (GSYNC Aj, R), the OB references consist
23 of *all* memory references issued by all participating processors before the Gsync
24 instruction, and the OA references consist of all *scalar stores* issued by all
25 participating processors after the Gsync instruction. Thus, this instruction ensures
26 that all previous memory references are complete before any subsequent scalar stores
27 are performed.

28 For the CPU Gsync instruction (GSYNC Aj, CPU), the OB and OA
29 references are the same as for the regular Gsync instruction. In addition, the Gsync
30 Aj, CPU instruction causes instruction dispatch to cease until all current CPU activity

1 has completed. This means that all outstanding loads have completed, all
2 outstanding stores have become globally visible, and all currently active instructions,
3 both scalar and vector, have completed execution and written any architectural state.
4 This is a superset of the general Gsync instruction.

5 For the purpose of Gsync operation, atomic memory operations are
6 considered both scalar loads and stores. Thus, for all types of Gsync instruction,
7 AMOs are considered OB references if preceding the Gsync in program order, and
8 OA references if following the Gsync in program order.

9 Aj contains a right-justified, n-bit mask of participating processors, where n is
10 the number of processors in an MSP. Bit p is set if processor p is participating in the
11 barrier. Thus, a full barrier in a four-processor MSP would use a mask of 0xf. A
12 barrier between processors 0 and 2 would use the mask 0x5.

13 **Restrictions:**

14 All of the participating processors *must* execute a Gsync instruction, else the
15 Gsync will time out, causing a Gsync Timeout exception to be signaled at all
16 processors waiting at the Gsync. The value of the timeout mechanism is selectable
17 under OS control.

18 All participating processors must use the same type of Gsync (acquire,
19 release, etc.), else the behavior of the instruction is Undefined. All participating
20 processors must use the same Aj (mask) value, all bits of Aj not corresponding to a
21 participating processor must be zero and all bits of Aj corresponding to a
22 participating processor must be one, else the behavior of the instruction is Undefined
23 and a Sync Error exception will be signaled at all participating processors.

24 **Operation:**

25 See Description above.

26 **Exceptions:**

27 Sync Timeout, Sync Error

28 **Programming Notes:**

29 The regular Gsync (GSYNC Aj) instruction is typically used in conjunction
30 with multi-MSP synchronization mechanisms, such as critical sections and barriers.

1 A join_barrier() routine, for example, might perform a Gsync before joining the
2 barrier. This ensures that once the barrier has been satisfied, all other processors will
3 see the results of the MSPs work (specifically, the work of those processors that
4 participated in the Gsync).

5 The acquire Gsync (GSYNC Aj, A) instruction is typically used in
6 conjunction with a lock acquisition protecting a critical section. When an acquire
7 Gsync is issued between the access to a synchronization variable (the integer load)
8 and accesses to data protected by the variable, it guarantees that the synchronization
9 and data accesses occur in the correct order. However, the acquire Gsync does *not*
10 wait for stores before the Gsync to complete, which would unnecessarily delay entry
11 into the critical section. This is why the acquire Gsync is preferable to the general
12 Gsync for acquire events.

13 The release Gsync (GSYNC Aj, R) instruction is typically used in
14 conjunction with a lock release, barrier, or other synchronization event signalling to
15 other processors that the current MSP is done performing some task. When a Gsync
16 R is issued before writing a synchronization variable, it guarantees that any processor
17 observing the write to the synchronization variable will subsequently be able to
18 observe any work performed by the MSP before the release Gsync.

19 The release Gsync does *not* delay loads after the Gsync from executing until
20 the stores before the Gsync have become globally visible. This is why the release
21 Gsync is preferable to the general Gsync for release events.

22 The CPU Gsync (GSYNC Aj, CPU) should be used before context switches.
23 It can also be used in debugging to make exceptions more “precise.” Care must be
24 taken, however, to ensure that any Gsync with multiple bits set in the mask is
25 executed by all participating processors.

26

27 Implementation Note:

28 In some embodiments, an implementation does not necessarily have to
29 serialize the full execution of memory references before and after the Gsync. The
30 Gsync instruction requires, in some embodiments, only that the instructions appear to

occur in order. Thus, OA references could begin execution so long as their effect cannot be observed by any processor. A fetch for a write miss could be initiated, for example, before an earlier Gsync has completed. The implementation must be careful, however, not to violate the global ordering guarantees provided by the memory consistency model and the Gsync instruction.

Sync, MSP (MSYNC)

Table 2

31	26	25	20	19	14	13	8	7	6	5	0		
g						i		j		k		t	f
000010						000000				000000		00	
6						6		6		6		2	6

Formats: MSYNC Aj f = 010000 SCALAR
MSYNC Aj, P f = 010001
MSYNC Aj, V f = 010010

Purpose: Intra-MSP barrier and memory synchronization

Description:

The Msync instructions act as a barrier synchronization and memory ordering fence among any arbitrary set of single-stream processors within an MSP. Aj provides a mask indicating which processors of the MSP are participating in the barrier. All participating processors must issue Msyncs with consistent masks.

Each type of Msync identifies two sets of memory references by participating processors: ordered before (OB) and ordered after (OA) references. The OB references consist of some or all of the participating processors' memory references that *precede* the Msync in program order. The OA references consist of some or all of the participating processors' memory references that *follow* the Msync in program

1 order. The processors participating in an Msync may use different types of Msyncs,
2 which will simply determine which of each processor's memory references are
3 included in the set of OB and OA references.

4 The Msync guarantees that all of the OB references appear to occur before
5 any of the OA references, as observed by the processors participating in the Msync.
6 Specifically, no OB read will return the result from any OA write, no OB write will
7 overwrite the value of any OA write and all OA reads will observe all OB writes.
8 The Msync does *not* provide any ordering for memory references by participating
9 processors that do not fall into either the OB or OA category. Nor does it provide
10 any global ordering of operations as observed by non-participating processors.

11 For a processor issuing the regular Msync (MSYNC Aj) instruction, its OB
12 references consist of all memory references issued before the Msync instruction, and
13 its OA references consist of all memory references issued after the Msync
14 instruction. Thus, when all participants use the regular Msync, the operation
15 provides a complete ordering of memory references by participants preceding the
16 Msync and memory references by the participants following the Msync.

17 For a processor issuing the vector Msync (MSYNC Aj, V) instruction, its OB
18 references consist of all vector memory references before the Msync instruction, and
19 its OA references consist of all vector memory references issued after the Msync
20 instruction. Thus its scalar references are not ordered with respect to any other
21 references.

22 For a processor issuing the producer Msync (MSYNC Aj, P) instruction, its
23 OB references consist of all memory references issued before the Msync instruction,
24 and it has no OA references. Thus, its memory references after the Msync are not
25 ordered with respect to any other references.

26 Msync does not order AMOs with respect to other memory references.
27 Ordering of AMOs requires, in some embodiments, the use of Gsyncs.

28 Aj contains a right-justified, n-bit mask of participating processors, where n is
29 the number of processors in an MSP. Bit p is set if processor p is participating in the
30 barrier. Thus, a full barrier among a four-processor MSP would use a mask of 0xf.

1 A barrier between processors 0 and 2 would use the mask 0x5.

2 **Restrictions:**

3 All of the participating processors *must* execute an Msync instruction, else the
4 barrier will time out, causing an Sync Timeout exception to be signaled at all
5 processors waiting at the Msync. The value of the timeout mechanism is selectable
6 under OS control.

7 All participating processors must use the same Aj (mask) value, all bits of Aj
8 not corresponding to a participating processor must be zero and all bits of Aj
9 corresponding to a participating processor must be one, else the behavior of the
10 instruction is Undefined and a Sync Error exception will be signaled at all
11 participating processors.

12 **Exceptions:**

13 Sync Timeout, Sync Error

14 **Programming Notes:**

15 A regular Msync provides an effective barrier synchronization among the
16 participating processors. This mechanism can be used to “park” slave processors
17 while waiting for a master processor to execute a serial section and compute values
18 needed by the slaves. The slave processors will stall on the first memory request
19 after the Msync. When the master eventually executes its Msync, the slave processor
20 loads will complete and the slave processors will resume execution. Of course the
21 setting of the Msync time-out timer determines how long the slave processors can be
22 parked before causing a Msync Time-out exception. Since this is a precise
23 exception, the OS can restart a job if it determines that it is behaving properly.

24 The vector Msync should be used for protecting vector-vector and scalar-
25 vector dependencies among the processors. This can avoid performance penalties to
26 unrelated scalar references after the Msync.

27 The producer Msync should be used by the producer in producer-consumer
28 synchronization. This ensures that the consumer will see all previous memory
29 references of the producer, but avoids penalizing subsequent memory references by
30 the producer. Note that for the producer Msync to make sense, at least one processor

must use a regular or vector Msync. A synchronization event using nothing but producer Msyncs would have no functional effect at all.

Sync, Local Floating Point (LSYNC FP)

Table 3

31	26 25	20 19	14 13	8 7 6 5	0
g 000010	i 000000	j 000000	k 000000	t 00	f 001111
6	6	6	6	2	6

Format: LSYNC FP VECTOR

Purpose: Provide ordering with respect to previous floating point activity

Description:

The Lsync FP instruction causes instruction dispatch to cease until all previous scalar and vector floating point operations have completed, and their results are visible.

Restrictions:

None

Operation:

See Description above.

Exceptions:

None

Programming Notes:

Lsync FP can be used before reading the floating point status register to guarantee that the results of all previous floating point

instructions are visible. It can also be used to guarantee that any floating point exceptions from previous instructions will be generated before continuing with the instruction execution.

Sync, Local Instruction Stream (LSYNC FP)

Table 4

31	26	25	20	19	14	13	8	7	6	5	0		
g 000010			i 000000			j 000000			k 000000		t 00	f 001110	
6			6			6			6		2	6	

Format: LSYNC I VECTOR

Purpose:

Provide coherence between stores and instruction fetches

Description:

Lsync I, when used in conjunction with the Gsync instruction, causes subsequent instruction fetches to see the result of previous memory stores.

The results of previous scalar and vector stores by the *same* processor are visible to subsequent instruction fetches after performing a release Gsync (or regular or CPU Gsync) followed by an Lsync I.

The results of previous scalar and vector stores by a different processor within the same MSP are visible to subsequent instruction fetches after performing a release Gsync (or regular or CPU Gsync) that includes the writing processor, followed by an Lsync I. Msyncs are

1 insufficient for this purpose.

2 To ensure that instruction fetches by processor A will see the
3 results of stores by a processor B in a different MSP, the two processors
4 must first synchronize via Gsyncs. After processor B performs the
5 stores to processor A's instruction space, it must perform a release
6 Gsync (or regular or CPU Gsync) and then synchronize with processor
7 A (by writing a flag which processor A reads, for example). Processor
8 A must then perform an acquire Gsync (or regular or CPU Gsync) and
9 then perform an Lsync I. At this point, subsequent instruction fetches
10 by processor A will observe the stores performed by processor B.

11 **Restrictions:**

12 None

13 **Operation:**

14 See Description above.

15 **Exceptions:**

16 None

17 **Implementation Notes:**

18 On a typical implementation, this instruction will simply
19 invalidate the instruction cache, and then force subsequent instructions
20 to be re-fetched from the instruction cache, causing re-fills from the
21 memory hierarchy. The Gsyncs ensure that, even if instruction fetches
22 take a different path to the memory system than loads, the previous
23 stores to instruction space will be visible to the subsequent fetches.

24 **Programming Notes:**

25 This instruction can be used by de-buggers, by self-modifying
26 code, or when dynamically generating code.

Sync, Local Scalar-Vector (LSYNC S,V)

Table 5

31	26 25	20 19	14 13	8 7 6 5	0
g 000010	i 000000	j 000000	k 000000	t 00	f 001000
6	6	6	6	2	6

Format: LSYNC S,V VECTOR

Purpose:

Provide local memory ordering between scalar and vector references

Description:

Lsync S,V provides ordering between scalar and vector memory references issued by the same processor. All scalar loads and stores preceding the Lsync S,V in program order appear to execute before any vector loads or stores following the Lsync S,V in program order. Lsync S,V provides no ordering among multiple vector memory references. Nor does it ensure that preceding scalar references have been *globally* performed before subsequent vector references are performed. LSyncs have no effect on AMO ordering.

Restrictions:

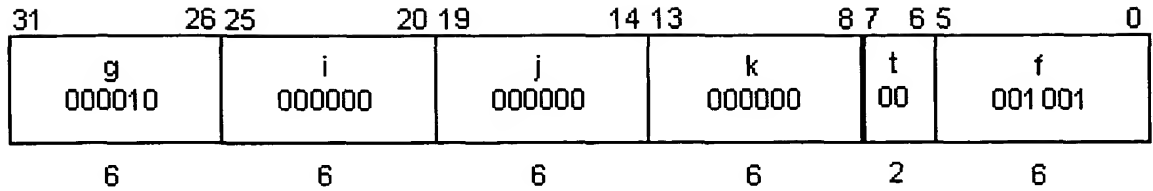
None

Operation:

See Description above.

Exceptions:

1	None
2	Programming Notes:
3	Scalar-vector Lsync instructions provide a lightweight mechanism
4	for ordering scalar memory references with subsequent vector
5	references.
6	Usage examples:
7	Scalar Store
8	LSYNC S,V ; protect against RAW hazard
9	Vector Load
10	...
11	Scalar Store
12	LSYNC S,V ; protect against WAW hazards
13	Vector Store
14	
15	Implementation Notes:
16	Note that Lsync S,V does not require prior scalar references to <i>be</i>
17	complete before allowing subsequent vector references to issue. It
18	simply requires, in some embodiments, that they must <i>appear</i> to have
19	completed. Thus, an implementation could simply ensure that
20	subsequent vector references have no way of passing prior scalar
21	references on the way to the cache and/or memory system.
22	
23	
24	Sync, Local Vector-Scalar (LSYNC V,S)
25	
26	Table 6



1

2

3 **Format:** LSYNC V,S VECTOR

4 **Purpose:**

5 Provide local memory ordering between vector and scalar
6 references

7 **Description:**

8 Lsync V,S provides ordering between vector and scalar memory
9 references issued by the same processor. All vector loads and stores
10 preceding the Lsync V,S in program order appear to execute before any
11 scalar loads or stores following the Lsync V,S in program order.

12 Lsync V,S provides no ordering among multiple vector memory
13 references. Nor does it ensure that preceding vector references have
14 been *globally* performed before subsequent scalar references are
15 performed. LSyncs have no effect on AMO ordering.

16 **Restrictions:**

17 None

18 **Operation:**

19 See Description above.

20 **Exceptions:**

21 None

22 **Programming Notes:**

23 Vector-scalar Lsync instructions provide a lightweight

1 mechanism for ordering vector memory references with subsequent
2 scalar references.

3 Usage example:

4 **Vector Store**
5 **LSYNC V, S ; protect against RAW hazard**
6 **Scalar Load**

7

8 **Implementation Notes:**

9 Note that Lsync V,S does not require prior vector references to *be*
10 complete before allowing subsequent scalar references to issue. It
11 simply requires, in some embodiments, that they must *appear* to have
12 completed. Thus, an implementation could simply ensure that
13 subsequent scalar references have no way of passing prior vector
14 references on the way to the cache and/or memory system.

15

16

17 **Sync, Local Vector-Scalar Read (LSYNC VR,S)**

18

19 Table 7

31	26 25	20 19	14 13	8 7 6 5	0
g 000010	i 000000	j 000000	k 000000	t 00	f 001010
6	6	6	6	2	6

20

21 **Format:** **LSYNC VR,S VECTOR**

22 **Purpose:**

23 Provide local memory ordering between vector loads and scalar

1 stores

2 **Description:**

3 Lsync VR,S provides ordering between vector loads and scalar
4 stores issued by the same processor. All vector loads preceding the
5 Lsync VR,S in program order appear to execute before any scalar loads
6 or stores following the Lsync VR,S in program order.

7 Lsync VR,S provides no ordering among multiple vector memory
8 references or between vector stores and scalar references. Nor does it
9 ensure that preceding vector loads have been *globally* performed before
10 subsequent scalar references are performed. LSyncs have no effect on
11 AMO ordering.

12 **Restrictions:**

13 None

14 **Operation:**

15 See Description above.

16 **Exceptions:**

17 None

18 **Programming Notes:**

19 Vector-scalar read Lsync instructions provide a lightweight
20 mechanism for providing WAR protection between vector loads and
21 subsequent scalar stores.

22 **Usage example:**

23 **Vector Load**

24 **LSYNC VR,S ; protect against WAR hazard**

25 **Scalar Store**

26

1 **Implementation Notes:**

2 Note that Lsync VR,S does not require prior vector loads to *be*
3 complete before allowing subsequent scalar references to issue. It
4 simply requires, in some embodiments, that they must *appear* to have
5 completed. Thus, an implementation could simply ensure that
6 subsequent scalar references have no way of passing prior vector loads
7 on the way to the cache and/or memory system. Note also that this
8 should generally be more lightweight than using the more restrictive
9 Lsync V,S as the scalar references do not have to wait for previous
10 vector stores to complete.

11 Implementations such as X1 that decouple store addresses from
12 store data remove the performance penalty of waiting for previous
13 vector stores. Thus, they may choose to simply interpret and implement
14 Lsync VR,S instructions as Lsync V,S instructions.

15
16

17 **Sync, Local Vector-Vector (LSYNC V,V)**

18
19

Table 8

31	26 25	20 19	14 13	8 7 6 5	0
g 000010	i 000000	j 000000	k 000000	t 00	f 001011
6	6	6	6	2	6

20

21 **Format: LSYNC V,V VECTOR**

22 **Purpose:**

23 Provide local memory ordering between groups of vector

1 references

2 **Description:**

3 Lsync V,V provides ordering between vector memory references
4 issued by the same processor. All vector loads and stores preceding the
5 Lsync V,V in program order appear to execute before any vector loads
6 or stores following the Lsync V,V in program order.

7 Lsync V,V provides no ordering between scalar and vector memory
8 references. Nor does it ensure that preceding vector references have
9 been *globally* performed before subsequent vector references are
10 performed. LSyncs have no effect on AMO ordering.

11 **Restrictions:**

12 None

13 **Operation:**

14 See Description above.

15 **Exceptions:**

16 None

17 **Programming Notes:**

18 Vector-vector Lsync instructions provide a lightweight
19 mechanism for ordering vector memory references, since they imply no
20 ordering with respect to scalar references.

21 **Usage examples:**

22 **Vector Store**

23 **LSYNC V,V ; protect against RAW hazard**

24 **Vector Load**

25

26 **Vector Store**

Vector Load

LSYNC V,V ; protect against WAR and WAW hazards

Vector Store

Implementation Notes:

Note that Lsync V,V does not require prior vector references to *be* complete before allowing subsequent vector references to issue. It simply requires, in some embodiments, that they must *appear* to have completed. Thus, an implementation could simply ensure that subsequent vector references have no way of passing prior vector references on the way to the cache and/or memory system.

Sync, Local Vector Register-Vector (LSYNC Vj,V)

Table 9

31	26 25	20 19 18	14 13	8 7	6 5	0
g 000010	i 000000	z 0	j	k 000000	t 00	f 001100
6	6	1	5	6	2	6

Format: LSYNC Vj,V VECTOR

Purpose:

Provide local memory ordering between specific vector references.

Description:

Lsync Vj,V provides ordering between specific vector memory

1 references issued by the same processor. The last memory reference
2 associated with vector register V_j preceding the
3 Lsync V_j, V in program order appears to execute before any vector loads
4 or stores following the Lsync V_j, V in program order.

5 Lsync V_j, V provides no ordering between scalar and vector
6 memory references, or between other (than the last associated with
7 vector register V_j) earlier vector references and subsequent vector
8 references. Nor does it ensure that the last vector reference associated
9 with register V_j has been *globally* performed before subsequent vector
10 references are performed. LSyncs have no effect on AMO ordering.

11 **Restrictions:**

12 None

13 **Operation:**

14 See Description above.

15 **Exceptions:**

16 None

17 **Programming Notes:**

18 Vector register-vector Lsync instructions provide a targeted
19 mechanism for ordering *specific* vector memory references, allowing the
20 memory pipeline to remain full with independent memory references. If
21 the load or store associated with register V_j has already completed
22 locally at the time that the Lsync V_j, V is issued, then the delay due to
23 synchronization can generally be minimized. Subsequent references do
24 *not* have to wait for all currently outstanding references to complete, just
25 the specified reference.

26 **Usage examples:**

1 **Vk Load ; load from array A into register Vk**
2 **[Other loads, ops and stores]**
3 **LSYNC Vj, V ; protect against WAR hazard involving**
4 **array A**
5 **Vector Store ; write to array A**
6

7 **Implementation Notes:**

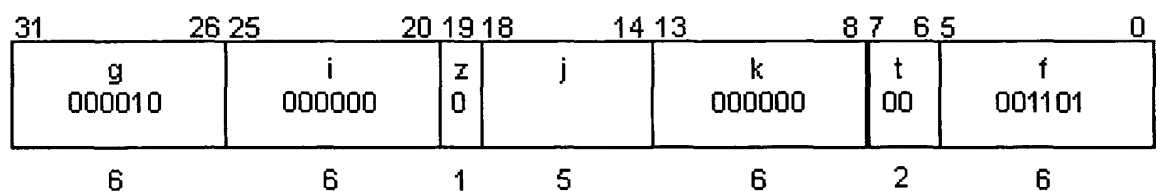
8 Note that Lsync Vj,V does not require the specified vector
9 reference to *be* complete before allowing subsequent vector references
10 to issue. It simply requires, in some embodiments, that it must *appear*
11 to have completed. Thus, an implementation could simply ensure that
12 subsequent references have no way of passing the specified reference on
13 the way to the cache and/or memory system.

14
15

16 **Sync, Local Vector Register-Vector Elemental (LSYNC V,V)**

17

18 Table 10



19

20 **Format: LSYNC Vj, V, EL VECTOR**

21 **Purpose:**

22 Provide local memory ordering for specific vector references by
23 elements

1 **Description:**

2 Lsync Vj,V,EL provides ordering between individual elements of
3 vector memory references issued by the same processor. The ordering is
4 provided between the last memory reference associated with vector
5 register Vj preceding the Lsync Vj,V,EL in program order, and all
6 vector loads or stores following the Lsync Vj,V,EL in program order.
7 Ordering is provided on a per-element basis, rather than on entire vector
8 references. That is, each element of the earlier reference appears to
9 execute before the corresponding (same number) element of any
10 subsequent vector reference.

11 Lsync Vj,V,EL provides no ordering between scalar and vector
12 memory references, between other (than the last associated with vector
13 register Vj) earlier vector references and subsequent vector references,
14 or between memory references from the local and other processors.
15 LSyncs have no effect on AMO ordering.

16 **Restrictions:**

17 None

18 **Operation:**

19 See Description above.

20 **Exceptions:**

21 None

22 **Programming Notes:**

23 Elemental vector Lsync instructions are most useful for protecting
24 against intra-loop anti-dependencies in the same iteration, arising from
25 code such as:

26 **for i = 1,n**

1 ... = ...A(i)...

2 A(i) = ...

3 In this case, the vector load from A and store to A can be
4 scheduled in the same chime, with an elemental vector Lsync protecting
5 against the WAR hazards on each element:

6 **Vi Load ; load from array A into Register Vk**

7 **[Other loads, ops and stores]**

8 **LSYNC Vj,V,EL ; protect against WAR hazard involving**
9 **array A**

10 **Vector Store ; write to array A**

11

12 Elemental vector Lsync instructions can also be used for
13 protecting against RAW hazards associated with a vector register spill
14 and re-fill. With 32 vector registers, however, spills should be relatively
15 rare.

16 **Implementation Notes:**

17 Note that Lsync Vj,V,EL does not require cross-pipe
18 synchronization in a multi-pipe implementation. Therefore, even in
19 situations in which the potentially conflicting vector references are
20 scheduled in different chimes, the elemental Lsync can be expected to
21 be somewhat lighter-weight than a full vector register-vector Lsync
22 when it is applicable.

23

24

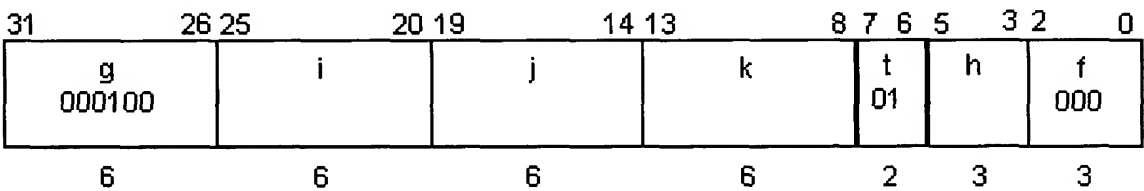
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

Atomic Memory Operations

In the assembler syntax shown; DL can be replaced with D or L. The symbol also can be omitted (be understood to stand for 64-bit operation when not present).

Atomic Fetch and Add

Table 11



Format: Ai,DL [Aj],AFADD,Ak,hint SCALAR

Purpose: Perform an atomic fetch-and-add on a memory location

Description:

The 64-bit signed integer value in Ak is added to the value in the memory location specified by Aj, and the old value of the memory location is returned into Ai. No integer overflow is detected. The read-modify-write sequence for the addition at memory is performed atomically with respect to any other atomic memory operations or stores.

All 64-bits of Ak are used and 64-bits of memory data are entered into Ai.

The hint can be na (non-allocate, h=2) or ex (cache exclusive, h=0). See for a description of the hint field values. The hint can be omitted, defaulting to na.

Restrictions:

The address in A_j must be naturally aligned. The bottom three bits must be zero, else an Address Error exception is signaled and A_i becomes Undefined.

Operand register numbers j and k must not differ by 32. If $\text{abs}(k-j) = 32$, then A_i becomes Undefined and an Illegal Register Reference exception is signaled.

The supported range of the virtual address is implementation specific. If the virtual address in A_j lies outside the supported range, then A_i becomes Undefined and an Address Error exception is signaled.

Operation:

$pAddr \leftarrow \text{AddressTranslation}(\text{ValueAR}(j, L), D)$

$memdata \leftarrow \text{LoadMemory}(D, pAddr, D)$

$\text{StoreAR}(i, L, memdata)$

$\text{StoreMemory}(D, (memdata + \text{ValueAR}(k, L)), pAddr)$

Exceptions:

TLB Miss, TLB Modify, TLB Conflict, Address Error, Illegal Register Reference

Programming Notes:

Note that atomic memory operations are not ordered by default with respect to scalar or vector loads or stores, nor are they ordered by Msyncs. Ordering of AMOs can be accomplished via Gsyncs or by true dependencies.

Atomic Compare and Swap

Table 12

31	26	25	20	19	14	13	8	7	6	5	3	2	0										
g 000100						i				j				k				t 01		h		f 010	
6						6				6				6				2		3		3	

Format: Ai,DL [Aj],ACSWAP,Ak SCALAR

Purpose: Perform an atomic masked swap on a memory location

Description:

The 64-bit integer value in Ai (the *comparand*) is compared to the value in the Doubleword memory location specified by Aj. If the values match, then the value in Ak (the *swaperand*) is stored into the memory location. In either case, the old value in the memory location is returned into Ai. The read-modify-write sequence for the swap is performed atomically with respect to any other atomic memory operations or stores.

All 64-bits of Ak are used and 64-bits of memory data are entered into Ai.

The hint can be na (non-allocate, h=2) or ex (cache exclusive, h=0). See for a description of the hint field values. The hint can be omitted, defaulting to na.

Restrictions:

The address in Aj must be naturally aligned. The bottom three bits must be zero, else an Address Error exception is signaled and Ai becomes Undefined.

Operand register numbers j and k must not differ by 32. If $\text{abs}(k-j) = 32$,

1 then A_i becomes Undefined and an Illegal Register Reference exception
2 is signaled.

3 The supported range of the virtual address is implementation specific. If
4 the virtual address in A_j lies outside the supported range, then A_i
5 becomes Undefined and an Address Error exception is signaled.

6 **Operation:**

```
7           pAddr  $\leftarrow$  AddressTranslation(ValueAR( $j$ ,  $L$ ),  $D$ )
8           temp  $\leftarrow$  LoadMemory( $D$ , pAddr,  $D$ )
9           if (temp = ValueAR( $i$ ,  $L$ )) then
10               StoreMemory( $D$ , ValueAR( $k$ ,  $D$ ), pAddr)
11           endif
12           StoreAR( $i$ ,  $L$ , temp)
```

13 **Exceptions:**

14 TLB Miss, TLB Modify, TLB Conflict, Address Error, Illegal
15 Register Reference

16 **Programming Notes:**

17 Note that atomic memory operations are not ordered by default
18 with respect to scalar or vector loads or stores, nor are they ordered by
19 Msyncs. Ordering of AMOs can be accomplished via Gsyncs or by true
20 dependencies.

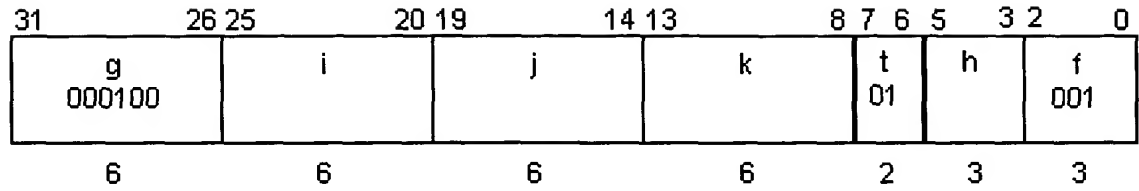
21

22

23

24 **Atomic Fetch and And-Exclusive Or**

25 Table 13



Format: Ai,DL [Aj],AFAX,Ak SCALAR

Purpose: Perform an atomic fetch and a logic function on a memory location

Description:

Under control of a 64-bit mask value in Ai, the Doubleword memory location specified by the memory byte address value in Aj, is modified using data from the 64-bit value in Ak. The read-modify-write sequence for the masked swap is performed atomically with respect to any other atomic memory operations or stores. The old value in the memory location, before modification, is returned into Ai.

This equation represents the logic function being performed on the specified memory location:

$$[Aj] \leftarrow (Ai \text{ and } [Aj]) \text{ xor } Ak$$

Each bit in the target memory Doubleword is ANDed with the corresponding bit of Ai. The result Doubleword is XORed with the corresponding bit of Ak. Using various combinations of Ai and Ak data bit values, masked store operations can be performed (store byte, for example) or logical operations can be performed on the specified memory Doubleword. AND, OR, XOR and EQV operations can all be performed as shown below.

All 64-bits of Ak and Ai are used and 64-bits of memory data are

1 entered into A_i .

2 The hint can be na (non-allocate, $h=2$) or ex (cache exclusive,
3 $h=0$). See for a description of the hint field values. The hint can be
4 omitted, defaulting to na.

5 **Restrictions:**

6 The address in A_j must be naturally aligned. The bottom three
7 bits must be zero, else an Address Error exception is signaled and A_i
8 becomes Undefined.

9 Operand register numbers j and k must not differ by 32. If $\text{abs}(k-$
10 $j) = 32$, then A_i becomes Undefined and an Illegal Register Reference
11 exception is signaled.

12 The supported range of the virtual address is implementation
13 specific. If the virtual address in A_j lies outside the supported range,
14 then A_i becomes Undefined and an Address Error exception is signaled.

15 **Operation:**

16 $pAddr \leftarrow \text{AddressTranslation}(\text{ValueAR}(j, L), D)$
17 $memval \leftarrow \text{LoadMemory}(D, pAddr)$
18 $newval \leftarrow (\text{ValueAR}(i, L) \text{ and } memval) \text{ xor } \text{ValueAR}(k, L)$
19 $\text{StoreMemory}(D, newval, pAddr)$
20 $\text{StoreAR}(i, L, memval)$

21 **Exceptions:**

22 TLB Miss, TLB Modify, TLB Conflict, Address Error, Illegal
23 Register Reference

24 **Programming Notes:**

25 The truth table of the logical function that is implemented by this
26 instruction is:

Table 14

Truth Table for Masked Logical Instructions

Ai	Ak	Prev Mem	New Memory	Result Ai (from Prev Mem)
0	0	0	0 (from Ak)	0
0	0	1	0 (Ak)	1
0	1	0	1 (Ak)	0
0	1	1	1 (Ak)	1
1	0	0	0 (Prev Mem)	0
1	0	1	1 (Prev Mem)	1
1	1	0	1 (Complement Mem)	0
1	1	1	0 (Complement Mem)	1

The following Table 15 shows how to set Ai and Ak to accomplish various logical functions to memory.

Table 15

Logical Operations for Masked Logical Instructions

Function	Ai	Ak	Memory Result in (Aj)
AND (Force clear on 0-bits)	operand	All zeros	(Aj) And operand
OR (Force set on 1-bits)	~operand	operand	(Aj) Or operand
XOR	All ones	operand	(Aj) Xor operand
EQV	All ones	~operand	(Aj) Eqv operand

1	Store under mask		operand, except	
2	(0 in mask stores	mask	0s where mask	Store operand
3	operand)		is 1s	under mask

Operand is the 64-bit operand value that will modify the specified memory location; the '~' symbol indicates taking the 1s complement.

Some examples:

To force the bottom 6 bits of a memory location to be 1s, leaving the other 58 bits in memory unchanged, set $A_i = 111...111000000$, A_k to $000...000111111$.

To store data into byte 1 (bits 15..8) of a memory location, leaving the other bits in the memory location unchanged, set $A_i = 111...111000000001111111$ and set A_k to $000...000ddddddd00000000$ where 'ddddddd' is the data byte being stored.

To set and clear bits in a memory location at the same time, something like this can be done:

$A_1 \leftarrow$ bits to be set

$A_2 \leftarrow$ bits to be cleared

$A_3 \leftarrow A_1 | A_2$

$A_3 \leftarrow \sim A_3$

$A_3 \leftarrow [A_4], AFAX, A_1$

Note that atomic memory operations are not ordered by default with respect to scalar or vector loads or stores, nor are they ordered by Msyncs. Ordering of AMOs can be accomplished via Gsyncs or by true dependencies.

Atomic Add

Table 16

31	26 25	20 19	14 13	8 7 6 5	3 2	0
g 000100	i	j	k 000000	t 01	h	f 100
6	6	6	6	2	3	3

Format: [Aj] Ai,DL,AADD SCALAR

Purpose: Perform an atomic integer add to a memory location

Description:

The 64-bit signed integer value in Ai is added to the value in the Doubleword memory byte address specified by Aj. No integer overflow is detected. The read-modify-write sequence for the addition in memory is performed atomically with respect to any other atomic memory operations or stores. No result is returned from memory.

All 64-bits of Ai are used.

The hint can be na (non-allocate, h=2) or ex (cache exclusive, h=0). See for a description of the hint field values. The hint can be omitted, defaulting to na.

Restrictions:

The address in Aj must be naturally aligned. The bottom three bits must be zero, else an Address Error exception is signaled.

Operand register numbers j and k must not differ by 32. If $\text{abs}(k-j) = 32$, then Ai becomes Undefined and an Illegal Register Reference exception is signaled.

The supported range of the virtual address is implementation specific. If the virtual address in Aj lies outside the supported range, then an

1 Address Error exception is signaled.

2 **Operation:**

3 $pAddr \leftarrow \text{AddressTranslation}(\text{ValueAR}(j, L), D)$

4 $\text{temp} \leftarrow \text{LoadMemory}(D, pAddr, D)$

5 $\text{StoreMemory}(D, \text{ValueAR}(i, D) + \text{temp}, pAddr)$

6 **Exceptions:**

7 TLB Miss, TLB Modify, TLB Conflict, Address Error

8 **Programming Notes:**

9 This instruction is identical to the fetch and add (AFADD)
10 instruction, except that it does not return a result. Thus it does not tie up
11 a result register, and will likely allow more pipelining in most
12 implementations than will the AFADD version.

13 AADD should be used in situations where a value in memory must
14 be modified, but the old value is not needed, such as when updating a
15 shared counter.

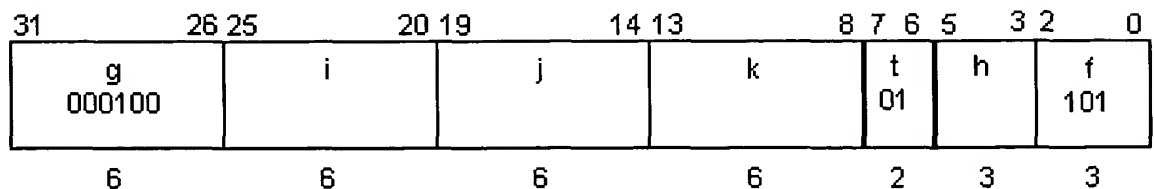
16 Note that atomic memory operations are not ordered by default with
17 respect to scalar or vector loads or stores, nor are they ordered by
18 Msyncs. Ordering of AMOs can be accomplished via Gsyncs or by true
19 dependencies.

20

21 **Atomic And-Exclusive Or**

22 Table 17

23



24

1 **Format:** [Aj] Ai,DL,AAX,Ak SCALAR

2 **Purpose:** Perform an atomic masked store on a memory location

3 **Description:**

4 Under control of a 64-bit mask value in Ai, the Doubleword memory
5 location specified by the memory byte address value in Aj, is modified
6 using data from the 64-bit value in Ak. The read-modify-write sequence
7 for the masked swap is performed atomically with respect to any other
8 atomic memory operations or stores. No result is returned from
9 memory.

10 This equation represents the logic function being performed on the
11 specified memory location:

12 $[Aj] \leftarrow (Ai \text{ and } [Aj]) \text{ xor } Ak$

13 Each bit in the target memory Doubleword is ANDed with the
14 corresponding bit of Ai. The result Doubleword is XORed with the
15 corresponding bit of Ak. Using various combinations of Ai and Ak data
16 bit values, masked store operations can be performed (store byte, for
17 example) or logical operations can be performed on the specified
18 memory Doubleword. AND, OR, XOR and EQV operations can all be
19 performed; see the Programming Notes for the instruction “Atomic
20 Fetch And-Exclusive Or.”

21 All 64-bits of Ak and Ai are used.

22 The hint can be na (non-allocate, h=2) or ex (cache exclusive, h=0).
23 See for a description of the hint field values. The hint can be omitted,
24 defaulting to na.

25 **Restrictions:**

26 The address in Aj must be naturally aligned. The bottom three bits

1 must be zero, else an Address Error exception is signaled.
2 Operand register numbers j and k must not differ by 32. If $\text{abs}(k-j) = 32$,
3 then A_i becomes Undefined and an Illegal Register Reference exception
4 is signaled.

5 The supported range of the virtual address is implementation specific. If
6 the virtual address in A_j lies outside the supported range, then an
7 Address Error exception is signaled.

8 **Operation:**

9 $\text{pAddr} \leftarrow \text{AddressTranslation}(\text{ValueAR}(j, L), D)$
10 $\text{memval} \leftarrow \text{LoadMemory}(D, \text{pAddr})$
11 $\text{newval} \leftarrow (\text{ValueAR}(i, L) \text{ and } \text{memval}) \text{ xor } \text{ValueAR}(k, L)$
12 $\text{StoreMemory}(D, \text{newval}, \text{pAddr})$

13 **Exceptions:**

14 TLB Miss, TLB Modify, TLB Conflict, Address Error, Illegal
15 Register Reference

16 **Programming Notes:**

17 See the Programming Notes for the instruction “Atomic Fetch and
18 And-Exclusive Or” for full details on functionality and use. This
19 instruction performs the same function except that the old memory
20 contents specified by A_j are not returned.

21 Note that atomic memory operations are not ordered by default with
22 respect to scalar or vector loads or stores, nor are they ordered by
23 Msyncs.

24

25 One aspect of the invention (e.g., useful for Lsync-type
26 operations), in some embodiments, provides an apparatus 200 or 500

1 that includes a first plurality of queues (e.g., 230 and 240 or 161 and
2 181), including a first queue and a second queue, each of the first
3 plurality of queues for holding a plurality of pending memory requests
4 237 and 247. The apparatus also includes, in some embodiments, one or
5 more instruction-processing circuits (e.g., IRQ 187 or VLSU 111),
6 operatively coupled to the plurality of queues, that inserts one or
7 memory requests into at least one of the queues based on a first
8 instruction that specifies a memory operation, and that inserts a first
9 synchronization marker into the first queue and inserts a second
10 synchronization marker into the second queue based on a second
11 instruction that specifies a synchronization operation, and that inserts
12 one or memory requests into at least one of the queues based on a third
13 instruction that specifies a memory operation. This apparatus also
14 includes a first synchronization circuit, operatively coupled to the first
15 plurality of queues, that selectively halts processing of further memory
16 requests from the first queue based on the first synchronization marker
17 reaching a predetermined point in the first queue until the corresponding
18 second synchronization marker reaches a predetermined point in the
19 second queue.

20 In some embodiments, the plurality of queues is within a
21 processor 100, and wherein the first queue is used for only vector
22 memory requests and synchronizations, and the second queue is used for
23 only scalar memory requests and synchronizations.

24 In some embodiments, the first synchronization instruction is an
25 Lsync-type instruction.

26 In some embodiments, the first synchronization instruction is an

1 Lsync V,S-type instruction.

2 In some embodiments, for a second synchronization instruction, a
3 corresponding synchronization marker is inserted to only to the first
4 queue.

5 In some embodiments, the second synchronization instruction is
6 an Lsync-type instruction.

7 Some embodiments further include a second plurality of queues,
8 including a third queue and a fourth queue, each of the second plurality
9 of queues for holding a plurality of pending memory requests, wherein
10 the plurality of queues is in a circuit coupled to each of a plurality of
11 processors including a first processor and a second processor, and
12 wherein the third queue is used for only memory requests and
13 synchronizations from the first processor, and the second queue is used
14 for only memory requests and synchronizations from the second
15 processor.

16 Some embodiments further include a second synchronization
17 circuit, operatively coupled to the second plurality of queues, that
18 selectively halts further processing of memory requests from the third
19 queue based on the first synchronization marker reaching a
20 predetermined point in the third queue until a corresponding
21 synchronization marker from the second processor reaches a
22 predetermined point in the fourth queue.

23 Some embodiments further include a fifth queue for holding a
24 plurality of write data elements, wherein each write data element
25 corresponds to a memory request in the first queue, and wherein the
26 write data elements are loaded into the fifth queue decoupled from the

1 loading of memory requests into the first queue.

2 Some embodiments further include a data cache, and an external
3 cache, wherein first synchronization instruction is an Lsync V,S type
4 instruction, preventing subsequent scalar references from accessing the
5 data cache until all vector references have been sent to the external
6 cache and all vector writes have caused any necessary invalidations of
7 the data cache.

8 Another aspect of the invention, in some embodiments, provides a
9 method that includes providing a first plurality of queues, including a
10 first queue and a second queue, each of the first plurality of queues for
11 holding a plurality of pending memory requests, inserting one or
12 memory requests into at least one of the queues based on a first
13 instruction that specifies a memory operation, based on a second
14 instruction that specifies a synchronization operation inserting a first
15 synchronization marker into the first queue and inserting a second
16 synchronization marker into the second queue, inserting one or memory
17 requests into at least one of the queues based on a third instruction that
18 specifies a memory operation, processing memory requests from the first
19 queue, and selectively halting further processing of memory requests
20 from the first queue based on the first synchronization marker reaching a
21 predetermined point in the first queue until the corresponding second
22 synchronization marker reaches a predetermined point in the second
23 queue.

24 In some embodiments of the method, the plurality of queues is
25 within a first processor, and wherein the inserting to first queue is for
26 only vector memory requests and synchronizations, and the inserting to

1 the second queue is for only scalar memory requests and
2 synchronizations.

3 In some embodiments of the method, the first synchronization
4 instruction is an Lsync-type instruction.

5 In some embodiments of the method, the first synchronization
6 instruction is an Lsync V,S-type instruction.

7 In some embodiments of the method, based on a second
8 synchronization instruction that specifies a synchronization operation,
9 inserting a corresponding synchronization marker to only the first
10 queue.

11 In some embodiments of the method, the second synchronization
12 instruction is an Lsync-type instruction.

13 Some embodiments of the method further include providing a
14 plurality of processors including a first processor and a second
15 processor, providing a second plurality of queues, including a third
16 queue and a fourth queue, each of the second plurality of queues for
17 holding a plurality of pending memory requests, inserting to the third
18 queue only memory requests and synchronizations from the first
19 processor, and inserting to the second queue only memory requests and
20 synchronizations from the second processor.

21 Some embodiments of the method further include selectively
22 halting further processing of memory requests from the third queue
23 based on the first synchronization marker reaching a predetermined
24 point in the third queue until a corresponding synchronization marker
25 from the second processor reaches a predetermined point in the fourth
26 queue.

1 Some embodiments of the method further include queueing a
2 plurality of write data elements to a fifth queue, wherein each write data
3 element corresponds to a memory request in the first queue, and wherein
4 the write data elements are inserted into the fifth queue decoupled from
5 the inserting of memory requests into the first queue.

6 Some embodiments of the method further include providing a
7 data cache and an external cache, wherein first synchronization
8 instruction is an Lsync V,S type instruction, and preventing subsequent
9 scalar references from accessing the data cache until all vector
10 references have been sent to the external cache and all vector writes
11 have caused any necessary invalidations of the data cache based on the
12 first synchronization instruction.

13 Another aspect of the invention (e.g., useful for Lsync-type
14 operations), in some embodiments, provides an apparatus that includes a
15 first plurality of queues, including a first queue and a second queue,
16 each of the first plurality of queues for holding a plurality of pending
17 memory requests, means as described herein for inserting one or
18 memory requests into at least one of the queues based on a first
19 instruction that specifies a memory operation, means as described herein
20 for, based on a second instruction that specifies a synchronization
21 operation inserting a first synchronization marker into the first queue
22 and inserting a second synchronization marker into the second queue,
23 means as described herein for inserting one or memory requests into at
24 least one of the queues based on a third instruction that specifies a
25 memory operation, means as described herein for processing memory
26 requests from the first queue, and means as described herein for

1 selectively halting further processing of memory requests from the first
2 queue based on the first synchronization marker reaching a
3 predetermined point in the first queue until the corresponding second
4 synchronization marker reaches a predetermined point in the second
5 queue.

6 In some embodiments, the plurality of queues is within a first
7 processor, and wherein the means for inserting to first queue operates
8 for only vector memory requests and synchronizations, and the means
9 for inserting to the second queue operates for only scalar memory
10 requests and synchronizations.

11 In some embodiments, the first synchronization instruction is an
12 Lsync-type instruction.

13 Another aspect of the invention (e.g., useful for Msync-type
14 operations), in some embodiments, provides an apparatus that includes a
15 first plurality of processors including a first processor and a second
16 processor, a first plurality of queues, including a first queue for holding
17 a plurality of pending memory requests from the first processor and a
18 second queue for holding a plurality of pending memory requests from
19 the second processor, wherein the first processor is operable to insert a
20 first msync marker into the first queue and the second processor is
21 operable to insert a second msync marker into the second queue, and a
22 synchronization circuit, operatively coupled to the first plurality of
23 queues, that halts processing of memory requests from the first queue
24 based on the first msync marker reaching a predetermined location in
25 the first queue until the corresponding second msync marker reaches a
26 predetermined location in the second queue.

1 In some embodiments, each msync marker includes a processor-
2 participation mask, and wherein the synchronization circuit checks the
3 processor participation mask of the first msync marker against the
4 processor participation mask in the second msync marker.

5 Some such embodiments further include a second plurality of
6 queues, including a third queue and a fourth queue, within the first
7 processor, and wherein the third queue is used for only vector memory
8 requests and synchronizations, and the fourth queue is used for only
9 scalar memory requests and synchronizations.

10 In some embodiments, the first msync marker is inserted into the
11 first queue based on a first synchronization instruction that is an Msync-
12 type instruction.

13 In some embodiments, the first msync marker is inserted into the
14 first queue based on a first synchronization instruction that is an Msync-
15 V-type instruction.

16 In some embodiments, for a second synchronization instruction, a
17 corresponding synchronization marker is inserted to only the third
18 queue.

19 In some embodiments, the second synchronization instruction is
20 an Msync-V-type instruction.

21 Some embodiments further include a second synchronization
22 circuit, operatively coupled to the second plurality of queues, that
23 selectively halts further processing of memory requests from the third
24 queue based on the first msync marker reaching a predetermined point
25 in the third queue until a corresponding msync marker reaches a
26 predetermined point in the fourth queue.

1 Some embodiments further include a fifth queue for holding a
2 plurality of write data elements, wherein each write data element
3 corresponds to a memory request in the first queue, and wherein the
4 write data elements are loaded into the fifth queue decoupled from the
5 loading of memory requests into the first queue.

6 Some embodiments further include a data cache internal to each
7 one of the first plurality of processors, and an external cache, wherein
8 the first msync marker is inserted into the first queue based on a first
9 synchronization instruction that is an Msync-type instruction, and is
10 held if the data cache in the first processor is currently in bypass mode
11 due to an earlier msync.

12 Another aspect of the invention, in some embodiments, provides a
13 method that includes providing a first plurality of processors including a
14 first processor and a second processor, providing a first plurality of
15 queues, including a first queue for holding a plurality of pending
16 memory requests from the first processor and a second queue for
17 holding a plurality of pending memory requests from the second
18 processor, inserting a first msync marker into the first queue from the
19 first processor, inserting a second msync marker into the second queue
20 from the second processor, and halting further processing of memory
21 requests from the first queue based on the first msync marker reaching a
22 predetermined location in the first queue until the corresponding second
23 msync marker reaches a predetermined location in the second queue.

24 In some embodiments, each msync marker includes a processor-
25 participation mask, and the method further includes checking the
26 processor participation mask of the first msync marker against the

1 processor participation mask in the second msync marker, and causing
2 special processing if the participation masks do not match.

3 Some embodiments further include providing a second plurality
4 of queues, including a third queue and a fourth queue, within the first
5 processor, inserting memory requests only if for vector accesses, and
6 appropriate msync markers to the third queue, and inserting memory
7 requests only if for scalar accesses, and appropriate msync markers to
8 the fourth queue.

9 In some embodiments, the inserting of the first msync marker into
10 the first queue based on a first synchronization instruction that is an
11 Msync-type instruction.

12 In some embodiments, the inserting of the first msync marker into
13 the first queue based on a first synchronization instruction that is an
14 Msync-V-type instruction.

15 Some embodiments further include, based a second
16 synchronization instruction, inserting a corresponding synchronization
17 marker to only the third queue.

18 In some embodiments, the second synchronization instruction is
19 an Msync-V-type instruction.

20 Some embodiments further include providing a second
21 synchronization circuit, operatively coupled to the second plurality of
22 queues, that selectively halts further processing of memory requests
23 from the third queue based on the first msync marker reaching a
24 predetermined point in the third queue until a corresponding msync
25 marker reaches a predetermined point in the fourth queue.

26 Some embodiments further include providing a fifth queue for

1 holding a plurality of write data elements, wherein each write data
2 element corresponds to a memory request in the first queue, and loading
3 the write data elements into the fifth queue in a manner that is decoupled
4 from the loading of memory requests into the first queue.

5 Some embodiments further include providing a data cache
6 internal to each one of the first plurality of processors, providing an
7 external cache, wherein the first msync marker is inserted into the first
8 queue based on a first synchronization instruction that is an Msync-type
9 instruction, and holding further processing if the data cache in the first
10 processor is currently in bypass mode due to an earlier msync.

11 Some embodiments further include preventing subsequent scalar
12 references in each processor from accessing the data cache until all
13 vector references have been sent to the external cache and all vector
14 writes have caused any necessary invalidations of the data cache.

15 Yet another aspect of the invention, in some embodiments,
16 provides an apparatus that includes a first plurality of processors
17 including a first processor and a second processor, a first plurality of
18 queues, including a first queue for holding a plurality of pending
19 memory requests from the first processor and a second queue for
20 holding a plurality of pending memory requests from the second
21 processor, means as described herein for inserting a first msync marker
22 into the first queue from the first processor, means as described herein
23 for inserting a second msync marker into the second queue from the
24 second processor, and means as described herein for halting further
25 processing of memory requests from the first queue based on the first
26 msync marker reaching a predetermined location in the first queue until

1 the corresponding second msync marker reaches a predetermined
2 location in the second queue.

3 Some embodiments further include means as described herein for
4 checking the processor participation mask of the first msync marker
5 against the processor participation mask in the second msync marker.

6 Some embodiments further include a second plurality of queues,
7 including a third queue and a fourth queue, within the first processor,
8 means as described herein for inserting memory requests but only if for
9 vector accesses to the third queue, and means as described herein for
10 inserting memory requests but only if for scalar accesses to the fourth
11 queue.

12 Yet another aspect of the invention (e.g., useful for Gsync-type
13 operations), in some embodiments, provides an apparatus that includes a
14 first plurality of processors including a first processor and a second
15 processor, a first plurality of queues, including a first queue for holding
16 a plurality of pending memory requests from the first processor and a
17 second queue for holding a plurality of pending memory requests from
18 the second processor, wherein the first processor is operable to insert a
19 first gsync marker into the first queue and the second processor is
20 operable to insert a second gsync marker into the second queue, and a
21 synchronization circuit, operatively coupled to the first plurality of
22 queues, that halts processing of memory requests from the first queue
23 based on the first gsync marker reaching a predetermined location in the
24 first queue until the corresponding second gsync marker reaches a
25 predetermined location in the second queue and until all outstanding
26 prior memory accesses have completed, whichever is later.

1 In some embodiments, each gsync marker includes a processor-
2 participation mask, and wherein the synchronization circuit checks the
3 processor participation mask of the first gsync marker against the
4 processor participation mask in the second gsync marker.

5 Some embodiments further include a second plurality of queues,
6 including a third queue and a fourth queue, within the first processor,
7 and wherein the third queue is used for only vector memory requests and
8 synchronizations, and the fourth queue is used for only scalar memory
9 requests and synchronizations.

10 In some embodiments, the first gsync marker is inserted into the
11 first queue based on a first synchronization instruction that is an Gsync-
12 type instruction. In some embodiments, for a second synchronization
13 instruction, a corresponding synchronization marker is inserted to only
14 the third queue.

15 In some embodiments, the second synchronization instruction is
16 an Gsync-V-type instruction.

17 Some embodiments further include a second synchronization
18 circuit, operatively coupled to the second plurality of queues, that
19 selectively halts further processing of memory requests from the third
20 queue based on the first gsync marker reaching a predetermined point in
21 the third queue until a corresponding gsync marker reaches a
22 predetermined point in the fourth queue.

23 Some embodiments further include a fifth queue for holding a
24 plurality of write data elements, wherein each write data element
25 corresponds to a memory request in the first queue, and wherein the
26 write data elements are loaded into the fifth queue decoupled from the

1 loading of memory requests into the first queue.

2 Some embodiments further include a data cache internal to each
3 one of the first plurality of processors, and an external cache, wherein
4 the first gsync marker is inserted into the first queue based on a first
5 synchronization instruction that is an Gsync-type instruction, and is held
6 if the data cache in the first processor is currently in bypass mode due to
7 an earlier gsync.

8 Yet another aspect of the invention (e.g., useful for Gsync-type
9 operations), in some embodiments, provides a method that includes
10 providing a first plurality of processors including a first processor and a
11 second processor, providing a first plurality of queues, including a first
12 queue for holding a plurality of pending memory requests from the first
13 processor and a second queue for holding a plurality of pending memory
14 requests from the second processor, inserting a first gsync marker into
15 the first queue from the first processor, inserting a second gsync marker
16 into the second queue from the second processor, and halting further
17 processing of memory requests from the first queue based on the first
18 gsync marker reaching a predetermined location in the first queue until
19 the corresponding second gsync marker reaches a predetermined
20 location in the second queue and until all outstanding prior memory
21 accesses have completed.

22 In some embodiments, each gsync marker includes a processor-
23 participation mask, and the method further includes checking the
24 processor participation mask of the first gsync marker against the
25 processor participation mask in the second gsync marker, and causing
26 special processing if the participation masks do not match.

1 Some embodiments further include providing a second plurality
2 of queues, including a third queue and a fourth queue, within the first
3 processor, inserting memory requests only if for vector accesses, and
4 appropriate gsync markers to the third queue, and inserting memory
5 requests only if for scalar accesses, and appropriate gsync markers to the
6 fourth queue.

7 In some embodiments, the inserting of the first gsync marker into
8 the first queue based on a first synchronization instruction that is an
9 Gsync-type instruction.

10 Some embodiments further include based a second
11 synchronization instruction, inserting a corresponding synchronization
12 marker to only the third queue.

13 In some embodiments, the second synchronization instruction is
14 an Gsync-V-type instruction.

15 Some embodiments further include providing a second
16 synchronization circuit, operatively coupled to the second plurality of
17 queues, that selectively halts further processing of memory requests
18 from the third queue based on the first gsync marker reaching a
19 predetermined point in the third queue until a corresponding gsync
20 marker reaches a predetermined point in the fourth queue.

21 Some embodiments further include providing a fifth queue for
22 holding a plurality of write data elements, wherein each write data
23 element corresponds to a memory request in the first queue, and loading
24 the write data elements into the fifth queue in a manner that is decoupled
25 from the loading of memory requests into the first queue.

26 Some embodiments further include providing a data cache

1 internal to each one of the first plurality of processors, providing an
2 external cache, wherein the first gsync marker is inserted into the first
3 queue based on a first synchronization instruction that is an Gsync-type
4 instruction, and holding further processing if the data cache in the first
5 processor is currently in bypass mode due to an earlier gsync.

6 Some embodiments further include preventing subsequent scalar
7 references in each processor from accessing the data cache until all
8 vector references have been sent to the external cache and all vector
9 writes have caused any necessary invalidations of the data cache.

10 Yet another aspect of the invention (e.g., useful for Gsync-type
11 operations), in some embodiments, provides an apparatus that includes a
12 first plurality of processors including a first processor and a second
13 processor, a first plurality of queues, including a first queue for holding
14 a plurality of pending memory requests from the first processor and a
15 second queue for holding a plurality of pending memory requests from
16 the second processor, means as described herein for inserting a first
17 gsync marker into the first queue from the first processor, means for
18 inserting a second gsync marker into the second queue from the second
19 processor, and means for halting further processing of memory requests
20 from the first queue based on the first gsync marker reaching a
21 predetermined location in the first queue until the corresponding second
22 gsync marker reaches a predetermined location in the second queue and
23 until all outstanding prior memory accesses have completed.

24 In some embodiments, each gsync marker includes a processor-
25 participation mask, and the apparatus further includes means for
26 checking the processor participation mask of the first gsync marker

1 against the processor participation mask in the second gsync marker.

2 Some embodiments further include a second plurality of queues,
3 including a third queue and a fourth queue, within the first processor,
4 means for inserting memory requests but only if for vector accesses to
5 the third queue, and means for inserting memory requests but only if for
6 scalar accesses to the fourth queue.

7 In the above discussion, the term “computer” is defined to include
8 any digital or analog data processing unit. Examples include any
9 personal computer, workstation, set top box, mainframe, server,
10 supercomputer, laptop or personal digital assistant capable of
11 embodying the inventions described herein.

12 Examples of articles comprising computer readable media are
13 floppy disks, hard drives, CD-ROM or DVD media or any other read-
14 write or read-only memory device or network connection.

15 Portions of the above description have been presented in terms of
16 algorithms and symbolic representations of operations on data bits
17 within a computer memory. These algorithmic descriptions and
18 representations are the ways used by those skilled in the data processing
19 arts to most effectively convey the substance of their work to others
20 skilled in the art. An algorithm is here, and generally, conceived to be a
21 self-consistent sequence of steps leading to a desired result. The steps
22 are those requiring physical manipulations of physical quantities.
23 Usually, though not necessarily, these quantities take the form of
24 electrical or magnetic signals capable of being stored, transferred,
25 combined, compared, and otherwise manipulated. It has proven
26 convenient at times, principally for reasons of common usage, to refer to

1 these signals as bits, values, elements, symbols, characters, terms,
2 numbers, or the like. It should be borne in mind, however, that all of
3 these and similar terms are to be associated with the appropriate
4 physical quantities and are merely convenient labels applied to these
5 quantities. Unless specifically stated otherwise as apparent from the
6 following discussions, terms such as “processing” or “computing” or
7 “calculating” or “determining” or “displaying” or the like, refer to the
8 action and processes of a computer system, or similar computing device,
9 that manipulates and transforms data represented as physical (e.g.,
10 electronic) quantities within the computer system’s registers and
11 memories into other data similarly represented as physical quantities
12 within the computer system memories or registers or other such
13 information storage, transmission or display devices.

14 It is understood that the above description is intended to be
15 illustrative, and not restrictive. Many other embodiments will be
16 apparent to those of skill in the art upon reviewing the above
17 description. The scope of the invention should, therefore, be determined
18 with reference to the appended claims, along with the full scope of
19 equivalents to which such claims are entitled. In the appended claims,
20 the terms “including” and “in which” are used as the plain-English
21 equivalents of the respective terms “comprising” and “wherein,”
22 respectively. Moreover, the terms “first,” “second,” and “third,” etc.,
23 are used merely as labels, and are not intended to impose numerical
24 requirements on their objects.